Università di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica
in Tecnologie Informatiche
Anno accademico 2005-2006

Tesi di Laurea:

Traduzione automatica di descrizioni Di servizi Web

Candidato:
Andrea Lorenzani

Relatore:
Prof. Antonio Brogi

Controrelatore:
Prof. Andrea Maggiolo Schettini

A mia madre,
che mi ha permesso di intraprendere questa strada
e a tutti coloro
che mi hanno sostenuto nel percorrerla

Ringraziamenti

Vorrei esprimere la mia più sincera gratitudine al relatore di questa tesi, il prof. Antonio Brogi, per avermi seguito nella stesura e per la disponibilità dimostrata.
Ringrazio inoltre il dottorando Razvan Andrei Popescu che ha supervisionato il progetto.
Ringrazio infine mia mamma, mio fratello, e tutti coloro che mi hanno sostenuto durante il corso dei miei studi.

Indice generale

Indice delle illustrazioni

Capitolo 1 - Introduzione

Il *Service Oriented Computing* (SOC) sta emergendo come un paradigma di calcolo nuovo e promettente, centrato sulla nozione di servizio come elemento fondamentale per sviluppare applicazioni software. Come discusso in [12], i servizi sono componenti che devono permettere una rapida composizione a basso costo di applicazioni distribuite.

I servizi sono offerti dai fornitori (*service provider*), che si occupano della loro implementazione e mantenimento, e rilasciano la loro descrizione. Le descrizioni di servizi sono usate per mostrarne funzionalità, comportamento e qualità, e dovrebbero fornire le basi per la loro scoperta (*discovery*), il collegamento dell'interfaccia all'indirizzo fisico (*binding*) e la possibilità di composizione con altri servizi (*composition*).

I servizi possiedono l'abilità di connettersi ad altri servizi per permettere il completamento delle transizioni complesse, come il controllo di un credito, l'ordine di prodotti o l'acquisto.

La natura neutrale della piattaforma utilizzata crea l'opportunità di costruire servizi compositi componendo quelli già esistenti, siano essi elementari o complessi, offerti tipicamente da fornitori diversi (si faccia riferimento per esempio a [13]).

Il modello dei servizi Web include tre ruoli: clienti, fornitori e registri, dove i fornitori pubblicano i loro servizi in registri che vengono poi interrogati dai clienti per scoprire quelli che reputano interessanti.

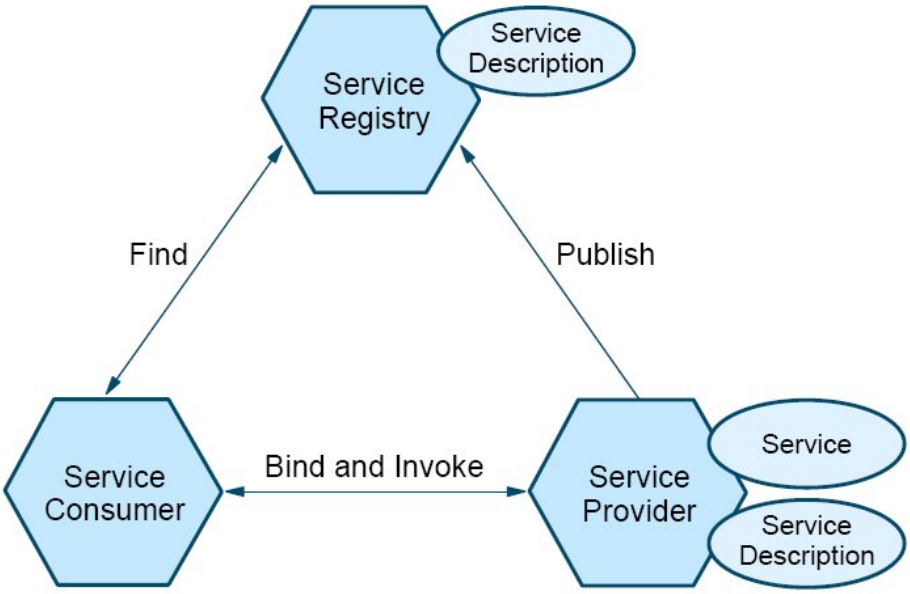


Figura 1.1: Service Oriented Architecture (SOA)

L'infrastruttura attuale dei Web service poggia su WSDL (*Web Services Description Language*), SOAP (*Simple Object Access Protocol*) e UDDI (*Universal Description & Discovery Interface*).

WSDL [15] è un linguaggio basato su XML per descrivere la funzione di un servizio e come invocarlo. SOAP [16] è un protocollo standard per scambiare messaggi tra applicazioni attraverso HTTP. UDDI [17] permette di definire registri globali dove vengono pubblicate le informazioni sui Web service. Attualmente UDDI è l'unico standard universalmente accettato per la scoperta di servizi Web.

BPEL4WS (*Business Process Execution Language for Web Services*) [1] è emerso come linguaggio per esprimere la composizione di Web Services. Un processo BPEL4WS fornisce il comportamento di un servizio Web in termini di coordinazione tra uno o più servizi WSDL.

Un aspetto negativo di BPEL4WS è che i clienti del processo di business sono incaricati di selezionare manualmente i servizi da comporre e di costruirsi il servizio composto.

Inoltre BPEL4WS non è provvisto di una semantica formale e quindi non fornisce mezzi adatti per un'analisi sistematica delle composizioni effettuate.

YAWL (*Yet Another Workflow Language*) è una proposta recente di un sistema di elaborazione di workflow/business che supporta un linguaggio di workflow conciso e potente e gestisce trasformazioni complesse di dati e integrazione di Web service. Questo è stato sviluppato da van der Aalst e ter Hofstede (si faccia riferimento a [2] e [3]) dopo una analisi dettagliata dei linguaggi di workflow esistenti. Per identificare i punti di forza e le carenze di questi linguaggi sono stati presi in considerazione i 20 pattern più comunemente usati nei workflow, ed è stata valutata la capacità di modellare questi pattern. Tra questi linguaggi quelli che si comportavano meglio erano quelli basati sulle reti di Petri. Tuttavia anche in questi casi non era facile modellare tutti i pattern presi in considerazione. Questo ha spinto i due autori di YAWL a sviluppare il nuovo linguaggio a partire dalle reti di Petri (ad alto livello) e aggiungendo meccanismi per permettere un supporto più diretto e intuitivo ai pattern identificati.

Ne è risultato un linguaggio con una semantica formale ben definita. La sua derivazione da una metodologia, come appunto quella delle reti di Petri, così ampiamente studiata permette di poter sfruttare tutta una serie di tecniche di verifica e convalida. A questo proposito è già stato sviluppato ad esempio un tool per la verifica di alcune proprietà [14].

Dato che implementa i pattern dei workflow più comuni, YAWL può inoltre essere usato come lingua franca per esprimere il comportamento dei Web service (descritti in linguaggi come BPEL4WS o OWL-S, per esempio), e può inoltre essere utilizzato per analizzare in maniera formale servizi reali.

1.1 Obiettivo della tesi e risultati ottenuti

Quello mostrato precedentemente è il contesto in cui si colloca il lavoro di questa tesi: il processo di esecuzione dei servizi Web può essere descritto in uno dei tanti linguaggi disponibili. Questo può portare a dei problemi nell'aggregare servizi,

soprattutto se consideriamo casi in cui tale aggregazione non viene fatta 'a mano' ma tramite aggregatori automatici.

L'obiettivo di questa tesi è quindi quello di ottenere una traduzione da BPEL4WS a YAWL che possa contribuire all'utilizzo di YAWL come linguaggio comune per l'aggregazione di servizi (si faccia riferimento a tal proposito a [10]) e di sviluppare un software che compia questa traduzione in maniera automatica, e che possa quindi essere utilizzato come componente per lo sviluppo di un aggregatore automatico.

A tal proposito vedremo che i vantaggi che otterremo saranno di avere una traduzione dettagliata di tutte le caratteristiche di BPEL4WS, anche quelle spesso non considerate in altri lavori (come *link* o gestori di eventi e errori, la cui descrizione viene data nel capitolo 2 di questa tesi) in un linguaggio emergente e potente come YAWL, che non è ancora mai stato considerato per traduzioni analoghe. In più tale traduzione verrà compiuta utilizzando un approccio composizionale, basato su pattern di traduzione istanziabili, e questo – oltre a consetire uno sviluppo disciplimato del software - potrà semplificare la futura progettazione di un traduttore inverso.

Verrà quindi descritta una implementazione in Java di tale traduttore che dimostrerà che la metodologia di traduzione può essere automatizzata, e quindi utilizzata veramente nell'ambito di un software integrato per l'aggregazione di servizi eterogenei..

Un ulteriore obiettivo di tale traduzione, infine, è quello di ottenere una specifica formale per il linguaggio BPEL4WS. Questo, in relazione con gli altri vantaggi appena discussi, permetterebbe la verifica delle caratteristiche dei servizi tramite software (già esistente [14] e in continua evoluzione) e quindi una migliore aggregazione.

Traduzioni analoghe a quella svolta in questa tesi hanno cercato di definire la semantica di BPEL4WS per mezzo di altre tecniche di formalizzazione. In [8] viene presentata una analisi di lavori simili compiuti in tale direzione, i cui risultati proncipali sono schematizzati nella seguente tabella:

	Technique	Structural Activities	Control links	Event and Fault Handlers
Fu et al. [18]	FSM	+	-	-
Foster et al. [19]	FSM	+	-	-
Fisteus et al. [20]	FSM	+	-	+/-
Ferrara [21]	PA	+	-	+
Koshkina & van Breugel [22]	PA	+	+	-
Farahbod et al. [23]	ASM	+	+/-	+
Martens[24] Hinz et al. [25]	PN	+	+/-	+
Stahl [26]	HPN	+	+	+
Ouyang et al. [8]	PN	+	+	+
Questa tesi	YAWL	+	+	+

Tabella 1.1: risultati ottenuti con lavori analoghi

Nella prima colonna sono riportati i progetti presi in esame. La colonna **Technique** indica la tecnica di formalizzazione usata (*FSM* per macchine a stati finiti, *PA* per algebre di processi, *ASM* per macchine a stati astratti, *PN* per le reti di Petri e *HPN* per le reti di Petri ad alto livello e la nostra che usa YAWL). Le colonne successive indicano quali caratteristiche vengono trattate dalle varie formalizzazioni. Qui il simbolo '+' indica che quel lavoro tratta quella caratteristica, '-' che non viene trattata, '+/-' che viene trattata ma solo parzialmente. La colonna **Structural Activities** indica se vengono tradotte le attività strutturate (per maggiori informazioni si rimanda al paragrafo *2.8*), e tutti i lavori presi in esame trattano almeno questa parte di BPEL4WS. **Control links** invece indica la gestione dei *control link*: come si vede dalla tabella più della metà dei lavori esaminati non considera questa caratteristica, o lo fa in maniera incompleta. Infine la colonna **Event and Fault Handlers** si riferisce alla gestione di eventi e fallimenti (descritti più avanti nel paragrafo *2.5.5*).

Come appena mostrato molti lavori non tengono in considerazione alcune parti della specifica BPEL4WS. La tecnica di traduzione proposta in questa tesi è basata su pattern diversi di rappresentazione delle singole attività BPEL4WS, e tratta *tutti* gli aspetti del linguaggio arrivando così a creare un workflow corrispondente che permetta una simulazione esatta del comportamento del Web Service. Come abbiamo detto, è stato quindi sviluppato un tool di traduzione automatica per trasformare i documenti

BPEL4WS in documenti YAWL.

Vale la pena osservare che la Tabella 1.1 indica che [26] e [8] sono due lavori che hanno trattato in maniera completa attività strutturate, control links e gestori di eventi e fallimenti di BPEL4WS. Una ovvia differenza di questa tesi rispetto a tali lavori è l'utilizzo di YAWL (anzichè delle reti di Petri) per la traduzione, e una conseguente rappresentazione più esplicita delle caratteristiche dei dati. Inoltre il supporto esplicito di pattern quali quelli di cancellazione (si veda il paragrafo 3.2 per maggior dettagli) permette una gestione più semplice di alcuni aspetti di BPEL4WS, come la terminazione dei processi.

1.2 Stesura della tesi

La tesi è stata divisa nei seguenti capitoli:

Capitolo 2: questo capitolo e il successivo si focalizzeranno sulle tecnologie alla base di questa tesi. Verrà condotta una analisi dettagliata di BPEL4WS prendendo in considerazione tutti gli aspetti del linguaggio, analisi che sarà finalizzata a comprendere ogni singolo problema riscontrato nella progettazione del traduttore e nella realizzazione del progetto;

Capitolo 3: si analizzerà in maniera approfondita il linguaggio YAWL, compreso il documento XML di scambio tra il programma editor di workflow e l'engine che permette la simulazione di un workflow, in quanto il tool di traduzione automatica genererà proprio un documento XML di questo tipo;

Capitolo 4: questo è il capitolo centrale della tesi. Si fornirà una panoramica della realizzazione del progetto di questa tesi. Verranno quindi mostrati in dettaglio nel paragrafo *4.1* gli schemi di traduzione delle singole *attività* BPEL4WS in YAWL, nel paragrafo *4.2* verranno mostrati i package creati per il progetto e le loro funzioni principali, e nel paragrafo *4.3* si mostrerà un esempio di funzionamento;

Capitolo 5: questa è la parte finale della tesi in cui saranno presenti alcune considerazioni critiche sul lavoro svolto e alcuni possibili sviluppi futuri.

Nell'ultima parte della tesi saranno raccolti nell'**Appendice A** tutti gli schemi di traduzione con le osservazioni.

Per permettere lo sviluppo di questo lavoro il codice è stato pubblicato come progetto su SourceForge (https://sourceforge.net/projects/bpel2yawl/).

Capitolo 2 - BPEL4WS (Business Process Execution Language for Web Services)

In questo capitolo verranno presi in esame i due linguaggi su cui si è svolto il progetto di tesi: BPEL4WS (Business Process Execution Language for Web Service) e YAWL (Yet Another Workflow Language), cercando di mettere in risalto le caratteristiche principali di questi due linguaggi.

2.1 Cenni alla nascita di BPEL4WS

La tecnologia di comunicazione ebbe come primo antenato il paradigma client-server. I computer clienti, attraverso una rete di comunicazione, interrogavano un computer servente adibito a elaborare risposte le quali, tramite lo stesso canale di comunicazione, raggiungevano nuovamente il cliente. Questo modo di scambiare informazioni portò alla creazione di tutta una serie di standard e di metodologie di comunicazione tra due singoli computer, un metodo di comunicazione molto utile ma inadatto alle varie necessità che pian piano stavano nascendo.

Per le imprese era importante risolvere il problema di poter fare interagire tra loro tutte le componenti interne e pian piano si creò l'esigenza di poter connettere direttamente tra loro i venditori, gli acquirenti e i fornitori attraverso l'infrastruttura preesistente.

Fu per questo che vennero studiate alcune soluzioni come ad esempio CORBA, che permette di realizzare la comunicazione tra client e server passando attraverso un canale comune chiamato ORB (Object Request Broker). Il canale comune permette ad applicazioni scritte in linguaggi diversi di poter scambiarsi la definizione delle proprie interfacce e, attraverso quelle, di interagire tra loro.

I *servizi web* (o *web service*) si sostituirono a soluzioni come CORBA nel tentativo di permettere interazioni tra differenti infrastrutture, facenti parte potenzialmente di diverse imprese, e resero più semplice la modifica delle componenti interne che costituiscono il servizio stesso. Comunque questa tecnologia aveva un evidente bisogno di parlare "*una stessa lingua*" per far sì che tutte le componenti potessero scambiarsi le informazioni necessarie, ed avere un metodo comune per impachettare i propri messaggi.

Venne così scelto come linguaggio per lo scambio dati l'XML (eXtensible Markup Language), mentre per la trasmissione su protocollo http e per impacchettare i messaggi si definì il SOAP (Simple Object Access Protocol).

Infine l'ultimo passo per fare dei *web service* una tecnologia matura era quello di avere un linguaggio di definizione dei *web service* stessi (tale linguaggio è il WSDL – Web Service Definition Language) e della loro orchestrazione.

WSDL è un linguaggio formale basato su XML che definisce l'interfaccia pubblica di un *web service*, ovvero descrive le informazioni essenziali per poter interagire con quello specifico servizio. Tali informazioni sono: le operazioni messe a disposizione, il protocollo di comunicazione con il formato dei messaggi scambiati e la sua ubicazione.

Il primo framework per la specifica dell'orchestrazione dei *web service* fu eCo, creato dalla CommerceNet, che dimostrava il valore dell'integrazione dei servizi di commercio elettronico, focalizzandosi sullo scambio di documenti per l'integrazione business-to-business (B2B). La specifica aveva una nozione vaga di orchestrazione che mostrava come un processo poteva essere composto da servizi web.

Il secondo lavoro che riguardava l'orchestrazione fu il WSCL (Web Service Conversation Language), che mostrava un semplice standard di conversazione, focalizzato sulla modellazione delle sequenze di interazioni tra *servizi web*.

In seguito la Microsoft sviluppò la specifica di XLANG per il Microsoft BizTalk Server. XLANG si focalizza sulla creazione di processi business e sull'iterazione tra provider di servizi web. La specifica fornisce supporto per flussi di controllo di processi sequenziali, paralleli e condizionali. Include inoltre una robusta gestione delle eccezioni, con supporto per transazioni di lunga durata tramite *compensazione*. XLANG usa WSDL per descrivere le interfacce di servizio di un processo.

Nel mentre la IBM proponeva il WSFL (Web Services Flow Language) per descrivere flussi per processi sia pubblici che privati. WSFL definisce uno specifico ordine di attività e scambio di dati per un processo particolare. Definisce inoltre sia la sequenza di esecuzione che la corrispondenza di ogni passo nel flusso su specifiche operazioni, riferite come modello del flusso e modello globale. Il modello del flusso rappresenta la serie di attività nel processo, mentre il modello globale collega ogni attività ad una specifica istanza di servizio web. Una definizione WSFL può essere inoltre esposta con una interfaccia WSDL, permettendo la decomposizione ricorsiva. Infine, WSFL supporta la gestione delle eccezioni ma non ha supporto diretto per le transazioni.

Queste ultime due specifiche, XLANG e WSFL, sono state superate da una nuova specifica creata dalla collaborazione della IBM, della Microsoft e della BEA. Questa specifica viene chiamata BPEL4WS (Business Process Execution Language for Web Service) e fa tesoro delle esperienze indipendenti delle specifiche create in precedenza, diventando, di fatto, il nuovo standard per l'orchestrazione dei *web service*. La versione a cui si fa riferimento in questa tesi è la 1.1, del 5 maggio 2003 (il cui riferimento è [1]).

2.2 Introduzione a BPEL4WS

Come abbiamo detto l'obiettivo dei *web service* è di ottenere una interoperabilità universale tra applicazioni usando gli standard della rete. I *web service* usano un modello che permette una integrazione flessibile di sistemi eterogenei in vari domini inclusi business-to-business e applicazioni aziendali.

Alla base di BPEL4WS sono presenti altre specifiche quali SOAP (che definisce un protocollo di messaggi XML per una interoperabilità basilare dei servizi), WSDL (che costituisce una grammatica comune per la descrizione dei servizi) e UDDI (che fornisce l'infrastruttura richiesta per la pubblicazione e la scoperta di servizi in maniera sistematica). Assieme queste specifiche permettono alle applicazioni di trovarsi tra di loro e di interagire seguendo un modello indipendente da piattaforma.

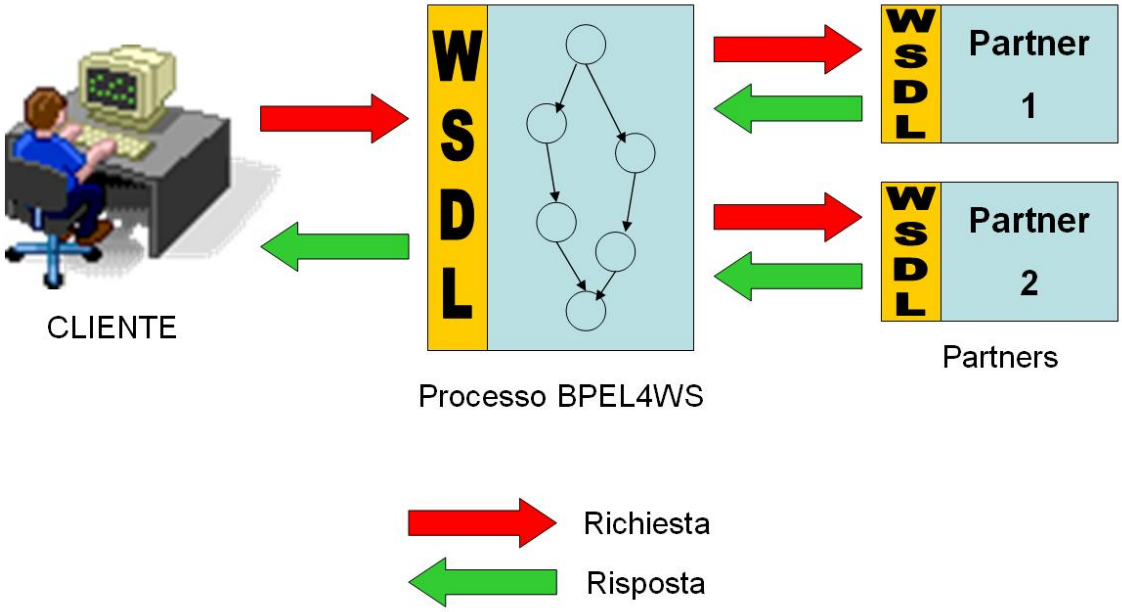


Figura 2.1: interazione con un web service e rapporto tra WSDL e BPEL4WS

Tuttavia l'integrazione dei sistemi richiede qualcosa di più che non l'abilità di condurre delle semplici interazioni usando protocolli standard: le applicazioni e i processi di business devono aver la possibilità di integrare le loro complesse interazioni usando un modello standard. Il modello fornito direttamente da WSDL è essenzialmente privo di stato con interazioni sincrone, oppure asincrone ma non correlate, mentre i modelli per le interazioni di business tipicamente assumono sequenze di scambio di messaggi alla pari, sia sincrone che asincrone, con interazioni che possiedono uno stato, hanno una durata estesa e coinvolgono più partner.

Questi protocolli di business richiedono quindi una specifica precisa dello scambio di messaggi tra i partecipanti, senza mostrare il comportamento interno del servizio. Ci sono due buone ragioni per separare gli aspetti esterni da quelli interni: la prima e più ovvia è legata alla segretezza che ogni partner vuole mantenere circa le decisioni interne, l'utilizzo dei propri dati e l'esecuzione delle proprie applicazioni; la seconda riguarda un lato più pratico, in quanto non mostrare indicazioni interne permette ai proprietari dei vari servizi di cambiare l'implementazione dei servizi stessi senza mettere in pericolo il corretto funzionamento di tutte le applicazioni correlate al servizio stesso.

I protocolli di business devono essere descritti in modo indipendente da piattaforma e devono catturare tutti gli aspetti comportamentali che abbiano un significato importante per più imprese. Gli utenti di un servizio devono solo comprendere e seguire il protocollo senza dover eseguire processi di accordo che aumentano la difficoltà nel creare applicazioni distribuite.

I concetti principali per descrivere un protocollo di business sono:

- comportamenti dipendenti dai dati (ad esempio, per un ordine ci possono essere comportamenti diversi legati al numero di oggetti ordinati o alla data di scadenza della spedizione) che richiedono l'uso di costrutti condizionali e di time-out
- possibilità di specificare condizioni eccezionali e loro conseguenze, comprese le sequenze di recupero
- interazioni a lungo termine che includono molteplici unità di lavoro spesso annidate, ognuna con la sua esigenza di dati

Per ottenere una descrizione precisa e predicibile del comportamento del servizio globale che può coinvolgere più imprese, è necessaria una ricca notazione, che avrà alcune funzionalità che ricorderanno un linguaggio di esecuzione.

Per quanto riguarda la gestione dei dati utilizzata da BPEL4WS, si usano proprietà del messaggio per identificare i dati rilevanti per il protocollo che sono racchiusi nei messaggi. Le proprietà possono essere viste come dati "trasparenti" utili per aspetti pubblici, contrapposti ai dati "opachi" usati per funzioni interne o private. Siccome sui dati "opachi" non si hanno informazioni e potrebbero portare a non determinismo, si assume che tutti i dati sensibili per un protocollo saranno messi all'interno di proprietà, e resi quindi trasparenti.

Tuttavia è possibile che una informazione scambiata e non apertamente definita in una proprietà possa cambiare il flusso dell'esecuzione di un servizio, come ad esempio potrebbe fare la richiesta di un ordine ad una libreria dove siano prenotati un numero di copie di un libro superiore a quelle disponibili. Per questo motivo è necessario avere qualcosa come un costrutto (in questo caso una *attività*) simile allo switch, in modo che

la scelta non sia deterministica al momento della definizione del servizio, ma che sia possibile determinare quali sono le possibilità che si possono manifestare.

I concetti base di BPEL4WS possono essere applicati in uno dei seguenti due modi:

- un processo BPEL4WS può definire un ruolo per un protocollo di business, usando la nozione di processo astratto. Per esempio in un servizio di ordini per una libreria vengono definiti due ruoli: venditore e acquirente. Ogni ruolo ha il suo processo astratto. Le relazioni tra i due sono modellate tipicamente tramite *partner link.* I processi astratti usano tutti i concetti di BPEL4WS ma hanno un approccio alla gestione dei dati tale da riflettere il livello di astrazione richiesto per descrivere gli aspetti pubblici del protocollo di business, ovvero gestiscono solo i dati relativi al protocollo, identificati come messaggi.
- è possibile usare BPEL4WS per definire un processo di business eseguibile. Logica e stato del processo determinano la natura e la sequenza delle interazioni condotte da ogni partner, e quindi il protocollo di interazione. Mentre per la definizione di un processo privato BPEL4WS non è richiesta la completezza, il linguaggio definisce effettivamente un formato di esecuzione portabile per i processi di business che si basa esclusivamente sulle risorse dei w*eb services* e sui dati XML.

Anche laddove aspetti di implementazione privati usano funzionalità dipendenti da piattaforma, la continuità del modello concettuale di base tra processo astratto ed eseguibile rende possibile esportare e importare aspetti pubblici incorporati nei protocolli di business come processi o template di ruoli mantenendo l'intento e la struttura dei protocolli. Questo aspetto è uno dei più attraenti perché permette lo sviluppo di strumenti e altre tecnologie che incrementano enormemente il livello di automatizzazione e in più diminuiscono il costo nello stabilire processi di business automatizzati tra imprese.

E' quindi utile che le due tipologie di utilizzo astratto ed eseguibile abbiano un nucleo comune di concetti per descrivere il processo.

In BPEL4WS viene quindi definito:

- un modello e una grammatica per descrivere il comportamento di un processo di business basato su interazioni tra il processo e i suoi partner, che avvengono attraverso le interfacce dei *web service*, la cui struttura di interazione è incapsulata nei *partnerLinks*
- come le interazioni multiple con questi partner sono coordinate per ottenere uno scopo, così come lo stato e la logica necessari alla coordinazione
- un meccanismo per gestire le eccezioni e i fallimenti che possono generarsi durante l'esecuzione di un processo
- un meccanismo per definire come alcune attività semplici o composte possano recuperare le informazioni gestite durante un processo che ha riscontrato una eccezione o un fallimento, in modo che si ristabilisca uno stato consistente e valido affinché l'eccezione o il fallimento riscontrato non compromettano ulteriormente l'esecuzione del processo (questo meccanismo prende il nome di *compensazione*)

Come già detto WSDL ha una grande influenza sul linguaggio BPEL4WS, che praticamente è costruito sopra questo, visto che al centro del modello di processo BPEL4WS c'è la nozione di interazione alla pari tra servizi descritti in WSDL: sia il processo che i partner sono modellati come servizi WSDL.

Un processo BPEL4WS definisce come coordinare le interazioni tra una sua istanza e i suoi partner e in questo senso la definizione del processo fornisce e/o usa uno o più servizi WSDL, e fornisce la descrizione del comportamento e delle interazioni di una istanza di processo relativa ai suoi partner e alle sue risorse attraverso interfacce di w*eb service*. Questo significa che BPEL4WS definisce il protocollo di scambio messaggi seguito dal processo di business di uno specifico ruolo nella interazione.

Infine BPEL4WS segue il modello di WSDL di separazione tra i contenuti dei messaggi astratti usati dal processo e le informazioni di deployment (messaggi e *portType* contro *binding* e informazioni di indirizzamento). In particolare BPEL4WS

rappresenta tutti i partner e le interazioni con questi in termini di interfacce astratte di WSDL: non viene fatto alcun riferimento al servizio attuale usato da una istanza di processo

2.3 Definizione WSDL per un processo BPEL4WS

Per spiegare come venga definito un processo tramite BPEL4WS verrà preso in esame l'esempio di servizio di gestione di un ordine che si può trovare in [1]. Il processo inizia con la ricezione di un messaggio di acquisto. Quando questo si verifica, vengono lanciate in parallelo tre operazioni: il calcolo del prezzo finale dell'ordine, la selezione del trasportatore e la programmazione della produzione dell'articolo e della sua consegna.

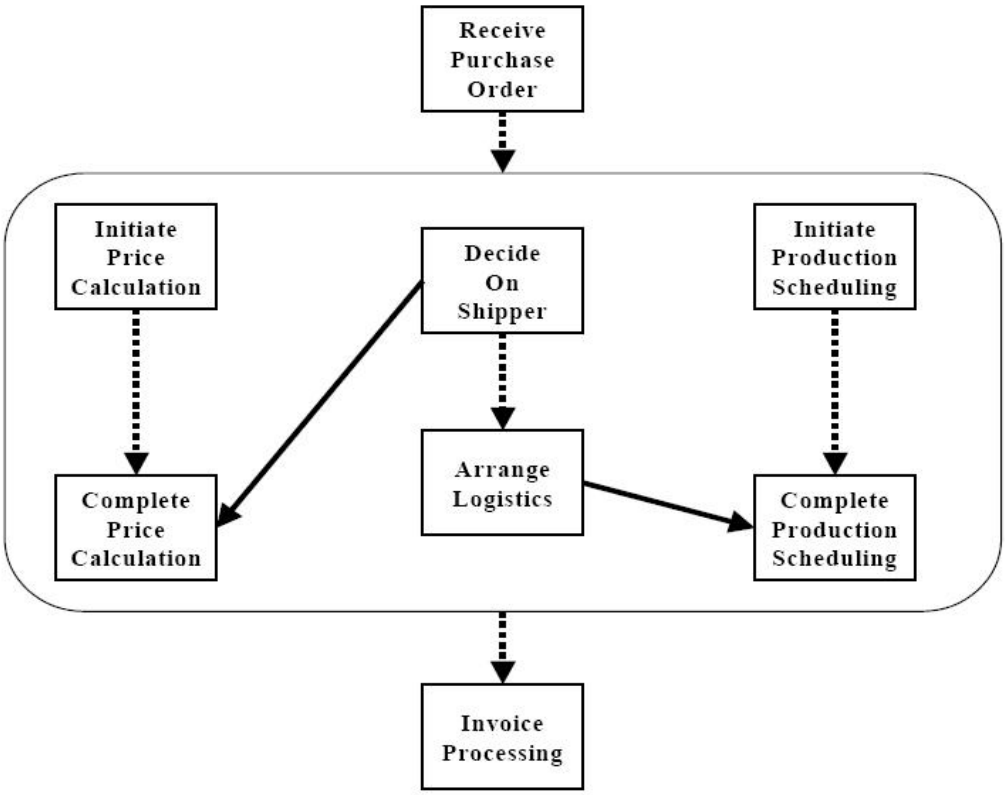


Figura 2.2: rappresentazione del servizio per l'esempio di BPEL4WS

Mentre alcuni passi possono essere fatti in maniera concorrente, ci sono controlli e dipendenze di dati tra i tre task. Per fare un esempio, è richiesto il prezzo di trasporto per calcolare il prezzo finale così come è richiesta la data di trasporto per programmare la consegna. Quando finalmente i tre task saranno eseguiti, si potrà procedere alla fatturazione, che verrà mandata all'acquirente.

Per esaminare a fondo tutti i dettagli della specifica del processo BPEL4WS sarà necessario analizzare anche il documento WSDL a cui tale processo sarà collegato, documento WSDL che conterrà tutte e sole le informazioni necessarie per la definizione del processo BPEL4WS. Questo vuol dire che in quel documento non si troveranno elementi di *binding* né elementi *service*, visto che il processo BPEL4WS è definito in maniera astratta facendo riferimento solo ai *portType* dei servizi coinvolti nel processo, e non al loro possibile deployment, permettendo quindi il riuso della definizione del processo con vari deployment di servizi compatibili.

Qui di seguito verranno brevemente analizzate le parti del documento WSDL a cui sarà associato l'esempio BPEL4WS.

```
  <definitions
targetNamespace="http://manufacturing.org/wsdl/purchase"
  xmlns:sns="http://manufacturing.org/xsd/purchase"
  xmlns:pos="http://manufacturing.org/wsdl/purchase"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  <import namespace="http://manufacturing.org/xsd/purchase"
  location="http://manufacturing.org/xsd/purchase.xsd"/>
```

Questa prima parte definisce i *namespace* a cui fa riferimento il documento WSDL, compreso quello di riferimento al documento stesso (*target namespace*). Essendo il documento WSDL scritto utilizzando XML è presente il *namespace* dello schema XML. Infine, utilizzando SOAP per la gestione dei messaggi, era necessario includere anche il

namespace di SOAP.

Definitions è l'elemento del documento WSDL che racchiude tutta la definizione del servizio.

Import permette di importare gli elementi e i tipi di uno schema XML in modo che possano essere riferiti all'interno del documento WSDL.

```
<message name="POMessage">
<part name="customerInfo" type="sns:customerInfo"/>
<part name="purchaseOrder" type="sns:purchaseOrder"/>
</message>
<message name="InvMessage">
<part name="IVC" type="sns:Invoice"/>
</message>
<message name="orderFaultType">
<part name="problemInfo" type="xsd:string"/>
</message>
<message name="shippingRequestMessage">
<part name="customerInfo" type="sns:customerInfo"/>
</message>
<message name="shippingInfoMessage">
<part name="shippingInfo" type="sns:shippingInfo"/>
</message>
<message name="scheduleMessage">
<part name="schedule" type="sns:scheduleInfo"/>
</message>
```

Questa seconda parte definisce i messaggi che verranno utilizzati, mandati e ricevuti dal servizio. I messaggi sono divisi in una o più parti logiche. Ogni parte è associata a un tipo di un qualche sistema di tipi. Ogni parte ha un nome per identificarla all'interno del messaggio.

Ovviamente anche i messaggi hanno un nome unico che li distingue tra di loro.

```
<portType name="purchaseOrderPT">
<operation name="sendPurchaseOrder">
<input message="pos:POMessage"/>
```

```
<output message="pos:InvMessage"/>
<fault name="cannotCompleteOrder"
message="pos:orderFaultType"/>
</operation>
</portType>
<portType name="invoiceCallbackPT">
<operation name="sendInvoice">
<input message="pos:InvMessage"/>
</operation>
</portType>
<portType name="shippingCallbackPT">
<operation name="sendSchedule">
<input message="pos:scheduleMessage"/>
</operation>
</portType>
```

Qui iniziano le definizioni dei *portType* usati dal servizio. Un *portType* è un insieme di operazioni astratte con i messaggi astratti ad esse associate. Anche in questo caso il nome del *portType* deve essere unico all'interno dello stesso documento WSDL. I messaggi di input e output vengono indicati mediante il loro nome.

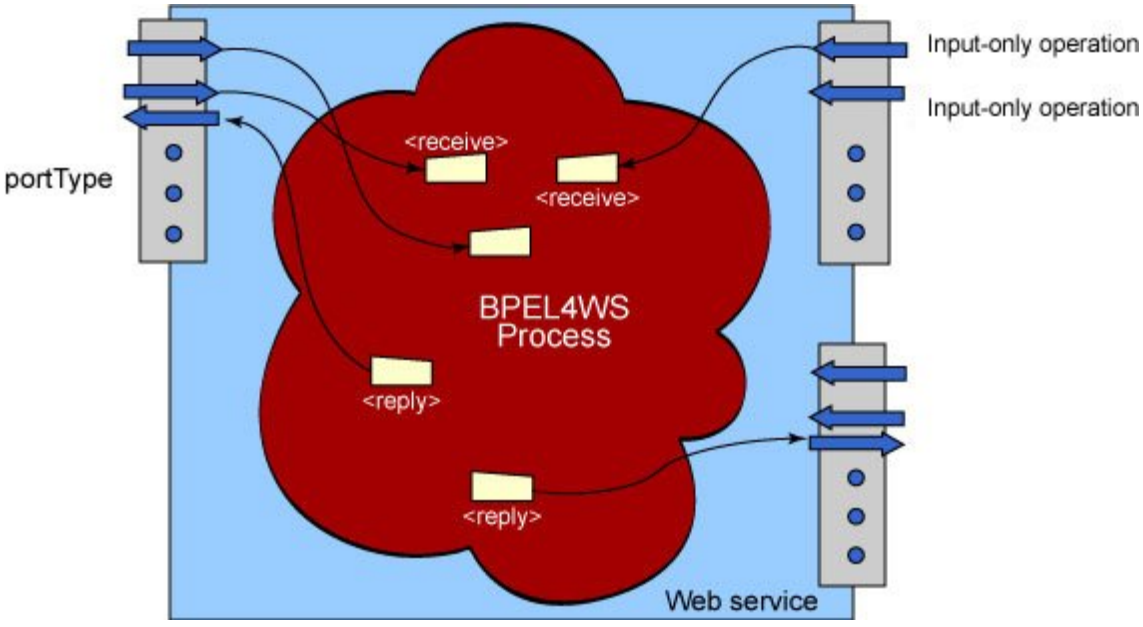


Figura 2.3: interazione tra servizio e portType esterni

WSDL ha quattro primitive di trasmissione che un *endpoint* (ovvero il servizio realmente in esecuzione) può utilizzare:

- unidirezionale: l'*endpoint* riceve un messaggio;
- richiesta-risposta: l'*endpoint* riceve un messaggio e restituisce una risposta;
- sollecito-risposta: l'*endpoint* invia un messaggio e riceve una risposta correlata;
- notifica: l'*endpoint* invia un messaggio.

Le operazioni specificate nei *portType* che definiscono solo il messaggio di input (nell'esempio **`invoiceCallbackPT`** e **`shippingCallbackPT`**) utilizzano come primitiva di trasmissione la unidirezionale, mentre quelle (nell'esempio solo la **`purchaseOrderPT`**) che definiscono input, output e messaggio di errore (fault) utilizzano la richiesta-risposta.

I *portType* sopra definiti si riferiscono solo al processo di ordine di acquisto. Di seguito invece saranno mostrati quelli utilizzati dal servizio di fatturazione, da quello di trasporto e da quello di programmazione della produzione

```
<!-- portType utilizzati dal servizio di fatturazione -->

<portType name="computePricePT">
<operation name="initiatePriceCalculation">
<input message="pos:POMessage"/>
</operation>
<operation name="sendShippingPrice">
<input message="pos:shippingInfoMessage"/>
</operation>
</portType>

<!-- portType utilizzati dal servizio di trasporto -->

<portType name="shippingPT">
```

```
<operation name="requestShipping">
<input message="pos:shippingRequestMessage"/>
<output message="pos:shippingInfoMessage"/>
<fault name="cannotCompleteOrder"
message="pos:orderFaultType"/>
</operation>
</portType>

<!-- portType supported by the production scheduling process -->

<portType name="schedulingPT">
<operation name="requestProductionScheduling">
<input message="pos:POMessage"/>
</operation>
<operation name="sendShipingSchedule">
<input message="pos:scheduleMessage"/>
</operation>
</portType>
```

Alla fine del documento WSDL sono specificati i *partnerLinkType* che rappresentano l'interazione tra il servizio di acquisto e tutti gli altri servizi con cui interagisce. Possono essere usati per rappresentare dipendenze tra servizi, senza doversi preoccupare del fatto che sia definito un processo per uno o più di questi servizi. Ognuno di questi *partnerLinkType* definisce fino a due nomi di ruoli, ed elenca i *portType* che ogni ruolo deve supportare per portare a termine l'interazione con successo.

Nell'esempio due *partnerLinkType* (**`purchasingLT`** e **`schedulingLT`**) elencano un solo ruolo perché, nelle corrispondenti interazioni del servizio, un solo partecipante fornisce tutte le operazioni invocate. **`purchasingLT`** rappresenta la connessione tra il processo e il cliente che ha fatto la richiesta di acquisto, e solo il servizio di ordine di acquisto deve offrire una operazione (**`sendPurchaseOrder`**).

`schedulingLT` invece rappresenta l'interazione tra il servizio di acquisto e quello di programmazione, in cui vengono invocate solo le operazioni del servizio di programmazione.

I rimanenti due *partnerLinkType* definiscono due ruoli perché sia il servizio di calcolo della fattura che quello di trasporto forniscono operazioni di ritorno per permettere che vengano spedite delle notifiche asincrone.

```
<plnk:partnerLinkType name="purchasingLT">
<plnk:role name="purchaseService">
<plnk:portType name="pos:purchaseOrderPT"/>
</plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="invoicingLT">
<plnk:role name="invoiceService">
<plnk:portType name="pos:computePricePT"/>
</plnk:role>
<plnk:role name="invoiceRequester">
<plnk:portType name="pos:invoiceCallbackPT"/>
</plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="shippingLT">
<plnk:role name="shippingService">
<plnk:portType name="pos:shippingPT"/>
</plnk:role>
<plnk:role name="shippingRequester">
<plnk:portType name="pos:shippingCallbackPT"/>
</plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="schedulingLT">
<plnk:role name="schedulingService">
<plnk:portType name="pos:schedulingPT"/>
</plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Per quanto riguarda una definizione WSDL completa di un servizio web, questa richiederebbe anche la specifica concreta del servizio per mezzo degli elementi *binding* (che specifica il protocollo concreto e il formato dei dati per le operazioni e per i

messaggi di uno specifico *portType*), *port* (che specifica un indirizzo per un *binding*, di fatto fornendo la locazione di un *endpoint* di comunicazione) e *service* (che è usato per aggregare un insieme di *port* correlate). Tuttavia queste parti del documento non sono necessarie per la definizione di un processo BPEL4WS.

2.4 Definizione di un processo BPEL4WS

Verrà adesso analizzato il processo BPEL4WS per il servizio di ordinazione mostrato nella *figura 2.2*. Nella definizione sono presenti quattro sezioni principali:

- la sezione dedicata ai *partnerLink* definisce i differenti partecipanti che interagiscono all'interno del processo di business nel corso della gestione dell'ordine. I quattro *partner* qui definiti corrispondono al mittente dell'ordine (*customer*), al fornitore del prezzo (*invoicingProvider*), al fornitore del trasporto (*shippingProvider*) e al servizio di programmazione della produzione (*schedulingProvider*). Ogni *partnerLink* è caratterizzato da un *partnerLinkType* e un nome di un ruolo. Queste informazioni identificano le funzionalità che devono essere supportate dal processo e dai partner del servizio affinché la relazione tra di loro abbia significato; ciò si traduce nel definire quali sono i *portType* che devono implementare il processo di ordine di acquisto e i partner di questo.
- La sezione delle variabili, che definisce le variabili per i dati usati nel processo, fornendone una definizione in termini di tipo di messaggi WSDL, tipo semplice in *XML-Schema* o elemento XML. Le variabili permettono di mantenere dati inerenti lo stato del processo e informazioni basate sullo scambio dei messaggi che riguardano l'esecuzione passata del processo stesso.
- La sezione di gestione degli errori contiene la definizione delle attività che devono essere intraprese in risposta ai fallimenti risultanti dalla invocazione dei servizi di valutazione e approvazione. In BPEL4WS tutti i

fallimenti, sia interni che risultanti da una invocazione di servizio, sono identificati da un nome. In particolare, ogni fallimento WSDL è identificato in BPEL4WS da un nome formato dal *namespace* del documento WSDL in cui sono definiti il portType e il fallimento in esame, e il nome del fallimento stesso (come appare nell'elemento *fault* dell'operazione). Da notare che la specifica WSDL 1.1 non richiedeva che i nomi di fallimenti fossero unici, quindi in caso di nomi identici non è possibile distinguere il fallimento generato. BPEL4WS fornisce un modello di denominazione uniforme per i fallimenti.

- Il resto della definizione del processo contiene la descrizione del normale comportamento per la gestione di una richiesta di acquisto. Gli elementi principali di questa parte della descrizione verranno spiegati nei paragrafi seguenti.

Andiamo ad esaminare brevemente la definizione BPEL4WS del processo:

```
<process name="purchaseOrderProcess"
targetNamespace="http://acme.com/ws-bp/purchase"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:lns="http://manufacturing.org/wsdl/purchase">
```

Come nel caso del documento WSDL, il documento si apre con la definizione dei *namespace* utilizzati e di quello principale che definisce il documento stesso. Al processo viene anche dato un nome.

```
<partnerLinks>
<partnerLink name="purchasing"
partnerLinkType="lns:purchasingLT"
myRole="purchaseService"/>
<partnerLink name="invoicing"
partnerLinkType="lns:invoicingLT"
myRole="invoiceRequester"
partnerRole="invoiceService"/>
```

```
<partnerLink name="shipping"
partnerLinkType="lns:shippingLT"
myRole="shippingRequester"
partnerRole="shippingService"/>
<partnerLink name="scheduling"
partnerLinkType="lns:schedulingLT"
partnerRole="schedulingService"/>
</partnerLinks>
```

Come precedentemente spiegato la prima sezione riguarda i *partnerLink*, e questi sono quelli definiti per il processo di acquisto, con i ruoli definiti per il processo stesso (*myrole*) o per il servizio partner (*partnerRole*). Ovviamente per i *partnerLinkType* si usa il *suffisso* del *namespace* del documento WSDL (in questo caso **lns**).

```
<variables>
<variable name="PO" messageType="lns:POMessage"/>
<variable name="Invoice" messageType="lns:InvMessage"/>
<variable name="POFault" messageType="lns:orderFaultType"/>
<variable name="shippingRequest"
messageType="lns:shippingRequestMessage"/>
<variable name="shippingInfo"
messageType="lns:shippingInfoMessage"/>
<variable name="shippingSchedule"
messageType="lns:scheduleMessage"/>
</variables>
```

Anche nel caso delle variabili le definizioni dei tipi di messaggio vanno ricercate nel documento WSDL, per questo motivo tutti i tipi di messaggio iniziano col *suffisso* (**lns**) del *namespace* del documento WSDL precedentemente specificato.

```
<faultHandlers>
<catch faultName="lns:cannotCompleteOrder" faultVariable="POFault">
<reply partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="sendPurchaseOrder"
variable="POFault"
```

```
faultName="cannotCompleteOrder"/>
</catch>
</faultHandlers>
```

Questo è l'unico gestore di fallimenti dichiarato per il processo. Per semplicità le operazioni generano soltanto un tipo di errore (**`lns:cannotCompleteOrder`**) che viene gestito inviando al cliente un messaggio di errore. Il significato dell'attività **`reply`** verrà spiegata in seguito. Qui basti sapere che tale attività definisce una risposta da inviare a un servizio che ha fatto una richiesta. Ovviamente per fare ciò occorre definire le informazioni importanti per questa risposta, quindi di che tipo è la variabile i cui dati verranno impachettati nella risposta (campo **`variable`**), oltre ai dati di definizione del partner (quindi **`partnerLink`**, **`portType`** e **`operation`**).

Quando si genera un fallimento il processo normale viene terminato e il controllo viene passato al corrispondente gestore definito nella sezione.

Qui di seguito è riportata la parte finale della definizione del processo BPEL4WS. La struttura del processo è formata da una sequenza di tre attività: la ricezione dell'ordine (**`receive`**), la sua gestione (**`flow`**) e la fatturazione (**`reply`**). La seconda attività tuttavia è più articolata e, come spiegato all'inizio, prevede l'esecuzione di tre parti in maniera concorrente (l'attività **`flow`** serve proprio a definire l'esecuzione in parallelo di più attività), ovvero il calcolo del prezzo, la decisione riguardo al servizio di trasporto e la programmazione della produzione. A loro volta queste tre attività sono sequenze di due attività ciascuna, con delle dipendenze tra di loro (per maggior chiarezza fare riferimento alla *figura 2.2*).

Nei paragrafi seguenti verranno analizzate tutte le attività che formano la specifica BPEL4WS.

```
<sequence>
<receive partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="sendPurchaseOrder"
variable="PO">
</receive>
```

```
<flow>
<links>
<link name="ship-to-invoice"/>
<link name="ship-to-scheduling"/>
</links>
<sequence>
<assign>
<copy>
<from variable="PO" part="customerInfo"/>
<to variable="shippingRequest"
part="customerInfo"/>
</copy>
</assign>
<invoke partnerLink="shipping"
portType="lns:shippingPT"
operation="requestShipping"
inputVariable="shippingRequest"
outputVariable="shippingInfo">
<source linkName="ship-to-invoice"/>
</invoke>
<receive partnerLink="shipping"
portType="lns:shippingCallbackPT"
operation="sendSchedule"
variable="shippingSchedule">
<source linkName="ship-to-scheduling"/>
</receive>
</sequence>
<sequence>
<invoke partnerLink="invoicing"
portType="lns:computePricePT"
operation="initiatePriceCalculation"
inputVariable="PO">
</invoke>
<invoke partnerLink="invoicing"
portType="lns:computePricePT"
operation="sendShippingPrice"
inputVariable="shippingInfo">
<target linkName="ship-to-invoice"/>
```

```
</invoke>
<receive partnerLink="invoicing"
portType="lns:invoiceCallbackPT"
operation="sendInvoice"
variable="Invoice"/>
</sequence>
<sequence>
<invoke partnerLink="scheduling"
portType="lns:schedulingPT"
operation="requestProductionScheduling"
inputVariable="PO">
</invoke>
<invoke partnerLink="scheduling"
portType="lns:schedulingPT"
operation="sendShippingSchedule"
inputVariable="shippingSchedule">
<target linkName="ship-to-scheduling"/>
</invoke>
</sequence>
</flow>
<reply partnerLink="purchasing"
portType="lns:purchaseOrderPT"
operation="sendPurchaseOrder"
variable="Invoice"/>
</sequence>
</process>
```

Il servizio visto assume che la richiesta di un cliente possa essere gestita in un breve arco di tempo, visto che il servizio si manifesta di tipo richiesta-risposta (una ricezione iniziale e una risposta finale). Per evitare questa assunzione si dovrebbe gestire questo scambio di messaggi in maniera unidirezionale mediante l'utilizzo di invocazioni (attività **`invoke`**) a interfacce di ascolto del cliente. Ovviamente questo richiederebbe la definizione di un secondo ruolo (il ruolo *cliente*) nel *partnerLinkType*, ruolo a cui compete l'ascolto dell'interfaccia di ricezione.

Per quanto riguarda le tre attività concorrenti, le dipendenze di sincronizzazione tra le attività interne alle tre sequenze sono espresse usando il costrutto **`link`** per collegarle.

Le attività stesse si dichiarano sorgente o destinazione di un **`link`**. Nel caso che una **`flow`** non contenga **`link`** le attività definite al suo interno procedono senza avere vincoli di sincronizzazione.

Nell'esempio sono presenti due soli **`link`** a definire le dipendenze di cui si è già parlato quando è stato introdotto l'esempio. Infatti, mentre il calcolo del prezzo può iniziare subito dopo la ricezione della richiesta, il prezzo del trasporto può essere aggiunto alla fatturazione solo dopo che questo valore sia stato ottenuto, e questa dipendenza tra la scelta del trasportatore e il calcolo del prezzo viene resa con la definizione del **`link`** chiamato **`ship-to-invoice`**.

Allo stesso modo il servizio di programmazione della produzione necessita di informazioni sui tempi di trasporto per poter indicare l'arrivo a destinazione del prodotto. Questa dipendenza viene resa con la definizione del **`link`** chiamato **`ship-to-scheduling`**.

Per quanto riguarda i dati, sono passati tra le differenti attività in maniera implicita attraverso la condivisione di variabili globali. Le dipendenze analizzate precedentemente sono legate a due variabili: **`shippingInfo`** che mantiene il costo del trasporto, quindi costituisce la variabile della dipendenza tra la scelta del trasportatore e il calcolo del costo totale (il **`link`** chiamato **`ship-to-invoice`**), e **`shippingSchedule`** che mantiene informazioni sui tempi di trasporto, usata per la dipendenza tra il servizio di trasporto e quello di programmazione della produzione (il **`link`** chiamato **`ship-to-scheduling`**).

I dati possono essere comunque copiati da una variabile all'altra mediante una attività di assegnamento (**`assign`**).

2.5 La struttura di un processo di business

In questo paragrafo scendiamo un po' più nei dettagli della sintassi di BPEL4WS. La struttura di base di un documento BPEL4WS è formata dalle seguenti parti:

- definizione di un processo coi suoi attributi ad alto livello;

- definizione dei partecipanti esterni al processo (*partnerLink* e *partner*)
- specifica dei dati scambiati mediante definizioni di variabili;
- definizione di *insiemi di correlazione* (*correlation set*);
- definizione dei gestori di errori, eventi e *compensazioni*;
- definizione delle attività che compongono il processo stesso;

di alcune di queste definizioni si è già discusso in precedenza. Tuttavia è importante, anche per l'esame del lavoro svolto per la tesi, spiegare la sintassi di BPEL4WS. Anche in questo caso si farà riferimento a [1] che è la specifica ufficiale di BPEL4WS.

2.5.1 Definizione del processo e attributi ad alto livello

```
<process name="ncname" targetNamespace="uri"
queryLanguage="anyURI"?
expressionLanguage="anyURI"?
suppressJoinFailure="yes|no"?
enableInstanceCompensation="yes|no"?
abstractProcess="yes|no"?
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
```

Questa è la prima parte della definizione del processo. La prima cosa che va notata è la simbologia: tutti i campi della sintassi usano una notazione tipica delle espressioni regolari, dove "**?**" indica la possibilità di non avere o avere unico l'elemento che precede tale simbolo, "**+**" indica la necessità di almeno un elemento di quel tipo e il simbolo "*****" la possibilità di non avere quell'elemento o di averlo in un numero arbitrario. L'assenza di simboli dopo un elemento indica che questo è obbligatorio e unico. Gli attributi di alto livello sono:

- **`name`** indica il nome del processo.
- **`targetNamespace`** definisce il *namespace* del documento BPEL.

- **`queryLanguage`** specifica il linguaggio usato per selezionare i nodi negli assegnamenti, nelle definizioni di proprietà e in altri usi. Il default è rappresentato dall'*XPath* 1.0.
- **`expressionLanguage`** definisce il linguaggio di espressione utilizzato all'interno del documento BPEL4WS. Anche in questo caso il default è costituito dal linguaggio *XPath* 1.0.
- **`suppressJoinFailure`** determina se l'errore **joinFault**, che si genera quando una condizione di join non viene verificata, sarà soppresso per tutte le attività del processo. Questo significa che in caso affermativo, generato un fallimento per una join non si procederà con la gestione dell'errore, ma si salterà l'attività la cui condizione di join non è stata verificata e si eliminerà il cammino dei link di cui l'attività era sorgente (questo fenomeno, chiamato *Dead-Path-Elimination*, verrà trattato di seguito). Il valore di default per questo attributo è "no", ovvero le condizioni di join non verificate generano un fallimento che deve essere gestito.
- **`enableInstanceCompensation`** determina se l'istanza di processo può essere compensata con significati specifici di una piattaforma. Il valore di default è "no".
- **`abstractProcess`** specifica se il processo è definito come *astratto* invece che *eseguibile*. Il valore di default è "no".

2.5.2 Definizione dei partecipanti esterni al processo

Definiti gli elementi di alto livello del processo, si definiscono i *partnerLinks*:

```
<partnerLinks>?
<!-- Nota: deve essere specificato almeno un ruolo. -->
<partnerLink name="ncname" partnerLinkType="qname"
myRole="ncname"? partnerRole="ncname"?>+
</partnerLink>
</partnerLinks>
```

Abbiamo già visto in precedenza il significato dei campi dei *partnerLinks*, che definiscono i partecipanti con cui interagisce il processo. Qui vengono riassunti brevemente gli attributi:

- **`name`** indica il nome, unico per ognuno dei partnerLink definiti, di questo specifico partecipante.
- **`partnerLinkType`** indica a quale *partnerLinkType* della definizione WSDL fa riferimento questo partecipante.
- **`myRole`** indica il ruolo, tra quelli definiti nella specifica WSDL, assunto dal processo nell'interazione con questo partecipante.
- **`partnerRole`** indica il ruolo, tra quelli definiti nella specifica WSDL, assunto nell'interazione dal partecipante esterno al processo.

```
<partners>?
<partner name="ncname">+
<partnerLink name="ncname"/>+
</partner>
</partners>
```

Questa parte della specifica è opzionale. Con la definizione di **`partners`** si intende porre dei vincoli su quali servizi deve aver disponibile un *partner* che interagirà con questo processo. Gli attributi richiesti sono:

- **`name`** che indica il nome del *partner* nell'interazione col processo.
- **`partnerLink name`** che indica il nome del *partnerLink* definito precedentemente che il partner dovrà fornire per prendere parte all'interazione col processo. Ovviamente un *partnerLink* non deve essere associato a due *partner* diversi, mentre è permesso che un *partnerLink* non sia associato ad alcun *partner*.

2.5.3 Definizione delle variabili

```
<variables>?
<variable name="ncname" messageType="qname"?
type="qname"? element="qname"?/>+
</variables>
```

Anche la definizione delle variabili è stata trattata in precedenza. Il significato dei campi viene qui riassunto brevemente:

- **`name`** specifica il nome unico tra le variabili assegnato a questa specifica variabile.
- **`messageType`** viene usato per specificare il tipo della variabile, che in questo caso sarà definito in un documento WSDL.
- **`type`** viene usato per specificare il tipo della variabile, che in questo caso sarà uno dei tipi semplici che sono definiti in *XML Schema*.
- **`element`** viene usato per specificare il tipo della variabile, che in questo caso sarà uno dei tipi complessi definiti tramite *XML Schema*.

Solo uno dei campi **`messageType`**, **`type`** e **`element`** deve essere presente nella definizione di una variabile di un processo BPEL4WS.

2.5.4 Definizione degli insiemi di correlazione

```
<correlationSets>?
<correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>
```

Questa parte definisce gli *insiemi di correlazione*. Per capire il significato di questi elementi bisogna sapere che WSDL di per se non ha una nozione di servizio con stato. Ma un processo ha bisogno di mantenere una correlazione tra i messaggi che invia e riceve e l'istanza del processo in esecuzione a cui questi sono riferiti. Per questo motivo si definisce un *insieme di correlazione*.

Il funzionamento a grandi linee non è complesso: due partecipanti ad una interazione devono semplicemente "accordarsi" su un elemento dei messaggi il cui valore permetta

di identificare l'istanza di processo a cui tali messaggi sono riferiti. Per fare un esempio nel caso precedentemente preso in esame del servizio di gestione degli ordini, un buon elemento per l'identificazione di una istanza potrebbe essere un numero di ordine associato ad ogni messaggio.

Il fatto che ogni servizio che partecipa ad una interazione possa aver bisogno di identificativi di istanza diversi può rendere il meccanismo delle correlazioni complicato. Non saranno presi in esame tutti i dettagli degli *insiemi di correlazione*. Basti sapere che le proprietà, ovvero i campi dei messaggi, che definiscono uno di questi insiemi vengono definite all'interno del documento WSDL in maniera simile a questa:

```
<bpws:property name="orderNumber" type="xsd:int"/>
```

dove il suffisso **bpws** indica il *namespace* di BPEL4WS, **name** indica il nome della *proprietà* e **type** indica il tipo di dato che costituisce la *proprietà* (nell'esempio è un intero definito come tipo semplice di *XML Schema*, il cui *suffisso* del *namespace* è appunto **xsd**).

Una volta definite le *proprietà* e i messaggi, all'interno del documento WSDL troveremo quindi una associazione tra una parte di un messaggio e la *proprietà*. Questa sarà definita in maniera simile alla seguente:

```
<bpws:propertyAlias propertyName="cor:orderNumber"
messageType="tns:POMessage" part="PO"
query="/PO/Order"/>
```

in questo esempio si sta indicando che la *proprietà* chiamata **cor:orderNumber** (quella definita prima) è associata al messaggio **tns:POMessage** (uno dei messaggi definiti nel documento WSDL), e, nello specifico messaggio, alla sua parte **PO** (vedere il paragrafo *2.3* per la definizione di una parte di un messaggio WSDL). Siccome quella parte è di tipo complesso, si usa una **query** scritta col linguaggio *XPath* (**/PO/Order** indica l'elemento di nome **Order** che costituisce **PO**) per indicare quale è l'intero a cui la *proprietà* si riferisce.

Terminate le definizioni, si specifica nella sezione dedicata ai *CorrelationSet* quali sono le *proprietà* per la correlazione tra messaggi e istanze, come nell'esempio:

```
properties="cor:customerID cor:orderNumber"
```

in cui questa correlazione è definita da due proprietà, di cui una è quella precedentemente presa in esame.

2.5.5 Definizione dei gestori per eventi, fallimenti e compensazioni

Definite le correlazioni tra messaggi e istanze di processo, vengono definiti i gestori di eventi ed errori come segue:

```
<faultHandlers>?
<!--  Nota: ci deve essere almeno o un gestore o quello di default
-->
<catch faultName="qname"? faultVariable="ncname"?>*
activity
</catch>
<catchAll>?
activity
</catchAll>
</faultHandlers>
```

questa è la definizione di un gestore di errori. Ogni elemento **`catch`** definisce come gestire uno specifico fallimento. Il fallimento gestito è specificato a partire dal suo nome (attributo **`faultName`**) come verrà spiegato di seguito.

Il gestore **`catchAll`** invece viene utilizzato come default per gli errori che non hanno un altro gestore specifico **`catch`** associato.

La gestione di un errore, quando questo si verifica, inizia con la terminazione di tutte le attività che fanno parte dello **`scope`** in cui si è generata l'eccezione.

A volte succede che, per gestire degli errori, sia necessario avere delle informazioni aggiuntive. Queste informazioni saranno contenute nella variabile definita dall'attributo

faultVariable. L'uso di questa variabile è legato al gestore, e la **faultVariable** per tale motivo non può essere riferita al di fuori di questo elemento.

Quando siamo in presenza di un errore che viene invocato con una variabile ad esso associata si controlla se è stato definito un gestore con tale **faultName** e con una variabile come **faultVariable** dello stesso tipo di quella associata all'errore. Se non ci sono gestori così definiti, si cerca un gestore che non abbia associato un **faultName** ma che abbia associato una variabile dello stesso tipo di quella che è stata associata all'errore. Se nemmeno in questo modo si trova un gestore adeguato, si ricorrerà al **catchAll**. Se **catchAll** non fosse definito e nessuna **catch** potesse gestire l'errore, nel caso di una **scope** l'errore verrebbe lanciato nella **scope** padre (che potrebbe essere il processo stesso), nel caso tale evento si verificasse invece nel processo allora questo terminerebbe in maniera anormale, come se venisse invocata l'attività di terminazione **terminate** (si veda a tal proposito il paragrafo *2.7.6*).

L'elemento ***activity*** definisce quali *attività* verranno svolte per gestire l'errore. Per una descrizione dettagliata delle attività si rimanda al paragrafo *2.6*.

```
<compensationHandler>?
activity
</compensationHandler>
```

Questo definisce come verrà gestita la *compensazione* del processo. Con questo termine si intende l'attività (o le attività) di correzione di una situazione collegata ad un errore. Ammettiamo per esempio che durante la gestione di un ordine di varie merci, una delle merci si scopre non essere più disponibile, mentre altre erano già state selezionate per l'ordine. La *compensazione* in questo caso potrebbe fare in modo che tutte le risorse già selezionate vengano rese nuovamente disponibili per un successivo ordine. In pratica la funzione della *compensazione* è quella di ripristinare uno stato consistente del processo in risposta ad un errore.

Proprio per il significato della *compensazione*, l'attività **compensate** che invoca in maniera esplicita il gestore di *compensazione* può essere definita solo all'interno di un gestore di errore o all'interno di un altro gestore di *compensazione* (questo perché

vedremo che è possibile definire gestori locali in attività chiamate **scope**).

Un gestore per uno **scope** viene installato, cioè ritenuto valido e utilizzabile in caso di *compensazione*, solo dopo che l'attività interna allo **scope** sia stata eseguita con successo. Se non esiste un gestore quando si verifica un errore, di default verranno chiamati tutti i gestori delle attività **scope** interni allo **scope** o al processo che ha generato l'errore, di cui sia già stato installato un gestore, nell'ordine inverso in cui queste attività sono state completate.

Se viene invocato un gestore che non è ancora stato installato, si eseguirà una operazione **empty** al suo posto. Non è definita la semantica di un processo in cui un gestore di *compensazione* che è stato installato viene invocato più di una volta.

```
<eventHandlers>?
<!-- Nota: ci deve essere almeno un gestore onMessage o onAlarm. -->
<onMessage partnerLink="ncname" portType="qname"
operation="ncname" variable="ncname"?>*
<correlations>?
<correlation set="ncname" initiate="yes|no"?>+
<correlations>
activity
</onMessage>
<onAlarm for="duration-expr"? until="deadline-expr"?>*
activity
</onAlarm>
</eventHandlers>
```

Questo definisce il gestore per gli eventi di ricezione di messaggi o di allarme. Ogni **onMessage** definisce un partecipante e i dati di ricezione di un messaggio che costituisce l'evento da gestire.

E' possibile associare ad un evento **onMessage** un *insieme di correlazione* (l'attributo **set** indica il nome dell'insieme a cui si fa riferimento) e indicare se, alla ricezione del messaggio, bisogna inizializzare (**initiate="yes"**) l'insieme di correlazione con le proprietà del messaggio definite dall'insieme stesso.

Gli eventi di allarme hanno definito soltanto il periodo di tempo dopo il quale

l'allarme verrà invocato. Questo periodo può essere un momento preciso (campo **`until`**) come ad esempio le ore 16.00 di oggi, o un periodo di attesa (campo **`for`**) ad esempio 20 minuti.

`activity` definisce quale o quali attività devono essere intraprese per gestire gli eventi.

Un gestore di eventi globale viene collegato ad una istanza del processo, quindi non viene considerato finché l'istanza di processo non sarà creata. Un gestore di eventi per uno **`scope`** sarà attivato quando l'attività **`scope`** inizierà, e sarà disattivato quando questa terminerà, ovviamente lasciando la possibilità di terminare le gestioni degli eventi già iniziate.

Per concludere la definizione di un documento BPEL4WS non rimane che da definire l'attività, o le attività, che fanno parte del processo:

```
activity
</process>
```

2.6 Le attività di un processo BPEL4WS

Il campo ***`activity`*** di cui si parlava precedentemente indica una delle *attività* che possono essere utilizzate in BPEL4WS per specificare un processo. Le *attività semplici*, che saranno analizzate nei prossimi paragrafi, sono le seguenti:

- *Receive*: per ricevere un messaggio.
- *Reply*: per rispondere ad una richiesta.
- *Invoke*: per invocare un servizio esterno.
- *Assign*: per copiare il valore di una variabile.
- *Throw*: chiama esplicitamente un gestore degli errori.
- *Compensate**:*** chiama esplicitamente il gestore della *compensazione*.
- *Terminate*: termina l'esecuzione del processo.
- *Wait*: attende un determinato periodo di tempo.

- *Empty*: non fa nulla.

Vi sono poi alcune *attività strutturate*, cioè che sono composte da una o più *attività semplici*. Le *attività strutturate* sono le seguenti:

- *Sequence*: raggruppa delle *attività* che devono essere svolte in sequenza.
- *Flow*: raggruppa delle *attività* che devono essere eseguite in parallelo.
- *Switch*: esegue una delle sue *attività* interne in base a una condizione.
- *While*: ripete l'esecuzione della sua *attività* interna finché non si verifica una condizione.
- *Pick*: attende un evento tra uno dei messaggi o degli allarmi in essa definiti per eseguire l'*attività* associata a questo evento.
- *Scope*: definisce un sottoprocesso con i propri gestori, le proprie variabili e le attività che ne costituiscono il comportamento e l'esecuzione.

Ovunque si parli di *attività* nella specifica BPEL4WS si intende o una *attività semplice* o una *attività strutturata*. Infatti anche il processo viene definito come una singola *attività*, proprio perché questa può essere strutturata, e quindi complessa a piacere.

Nei seguenti paragrafi verranno analizzate tutte le *attività*.

2.6.1 Attributi ed elementi standard delle attività e significato dei link

Qui di seguito verrà trattata l'ultima parte della sintassi del documento BPEL4WS, e il significato e i campi di ogni *attività*. Alcuni di questi campi sono *attributi standard*, presenti quindi in tutti i costrutti delle *attività*, e possono essere definiti anche degli *elementi standard* interni a tutte le *attività*, i **`link`**, che servono per regolare le dipendenze tra *attività* che sono in esecuzione contemporaneamente.

Gli attributi standard, con la loro sintassi, sono:

```
name="ncname"?
joinCondition="bool-expr"?
```

```
suppressJoinFailure="yes|no"?
```

dove **`name`** è il nome dell'*attività* che verrà utilizzato come riferimento, **`joinCondition`** e **`suppressJoinFailure`** sono collegate al funzionamento dei **`link`** e quindi spiegate dopo aver introdotto questi elementi.

Gli *elementi standard* sono:

```
<target linkName="ncname"/>*
<source linkName="ncname" transitionCondition="bool-expr"?/>*
```

definiscono sorgente (*source*) e destinazione (*target*) dei **`link`**. I **`link`** permettono di stabilire delle dipendenze di sincronizzazione tra le *attività* di un flusso (**`flow`**, vedi paragrafo *2.8.2*). Una *attività* che sia destinazione di uno o più **`link`** dovrà attendere che le *attività* sorgenti terminino prima di avere la possibilità di essere eseguita. Quando le *attività* sorgenti terminano la propria esecuzione valutano l'espressione contenuta nel campo **`transitionCondition`** (l'espressione, se non viene indicata, di default è "**`true()`**" che in *XPath* indica l'espressione sempre vera) e trasmettono tale valore all'*attività* di destinazione come status del proprio **`link`**.

Questa *attività*, che può essere annidata in qualsiasi livello interno alle *attività strutturate*, avrà associata una espressione **`joinCondition`** che deve essere sempre composta da valori booleani o da valori degli status dei suoi **`link`** (ottenuti tramite la funzione **`bpws:getLinkStatus`**). Se non dovesse essere definita una espressione, quella di default controllerà che *almeno uno* dei **`link`** abbia come status il valore "**true**" (OR logico dei valori di tutti gli status dei **`link`**). In base al valore di questa espressione si hanno i due seguenti casi:

- se la condizione è verificata (il suo valore booleano è "**true**") l'*attività* destinazione può essere eseguita normalmente;
- se la condizione non è verificata (il suo valore booleano è "**false**") e il valore del campo **`suppressJoinFailure`** è **`no`** (come da default di quel

campo) verrà lanciata una eccezione che interromperà l'esecuzione del processo e dovrà essere gestita dal gestore adatto;

- se la condizione non è verificata (il suo valore booleano è "**false**") e il valore del campo **`suppressJoinFailure`** è **`yes`** il processo continua la sua esecuzione saltando l'*attività* e propagando un **false** come status di tutti i **`link`** di cui è sorgente (*Dead-Path-Elimination*).

Riassumendo vedremo che tutte le *attività* contenute in un **`flow`** sono generalmente pronte ad essere eseguite nel momento in cui il **`flow`** stesso entra in esecuzione. Tuttavia andranno in esecuzione solo quelle che non sono destinazione di alcun **`link`**. Tra queste alcune potrebbero essere sorgente di **`link`** e quindi, una volta terminata la loro esecuzione, valuteranno la loro **`transitionCondition`** e assegneranno questo valore come status del **`link`**. Quelle che invece sono destinazione di qualche **`link`** dovranno attendere il completamento di tutte le *attività* loro sorgenti per poter valutare la **`joinCondition`** ad esse associata e quindi determinare se essere eseguite, saltate o se sollevare una eccezione. Se verranno eseguite valuteranno a loro volta il valore dei **`link`** di cui sono sorgente, se verranno saltate invece assegneranno il valore "**false**" ai **`link`** di cui sono sorgente, e a tutti i **`link`** di cui sono sorgenti le attività in essa contenute (*Dead-Path-Elimination*). E così via, fino a che tutte le *attività* interne al **`flow`** non saranno eseguite.

E' importante notare che i **`link`** possono non tenere conto dei confini delle *attività*, ovvero si possono collegare una *attività* A interna al **`flow`** con una *attività* B interna a una **`sequence`** interna a sua volta alla stessa **`flow`** (due livelli di annodamento diversi). Ci sono però delle limitazioni a questa regola, perché i **`link`** non possono varcare i confini di: *attività* **`while`**, **`scope`** serializzabili, un gestore di eventi o di *compensazione*. Possono varcare i confini di un gestore degli errori, ma solo se l'*attività* sorgente è interna al gestore e la destinataria è interna allo **`scope`** che racchiude lo **`scope`** associato al gestore.

2.7 Attività semplici

2.7.1 Receive e Reply

Un processo descrive un servizio web. Le *attività* che permettono di interagire con chi fa una richiesta al servizio stesso sono proprio la **`receive`** e la **`reply`**. La prima permette la ricezione di un messaggio, mentre la seconda permette di rispondere al mittente del messaggio di richiesta. Una **`receive`** seguita, dopo altre attività, da una **`reply`** si traduce quindi in una interazione domanda-risposta.

Il costrutto **`receive`** permette quindi al processo di bloccarsi in attesa di un messaggio in arrivo.

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
variable="ncname"? createInstance="yes|no"?
attributi standard>
elementi standard
<correlations>?
<correlation set="ncname" initiate="yes|no"?>+
</correlations>
</receive>
```

Una **`receive`** deve essere associata ad una interfaccia esterna. Questa viene definita attraverso gli attributi **`partnerLink`**, **`portType`** e **`operation`** che definiscono il ruolo del servizio in questa interazione, il *portType* WSDL utilizzato per la comunicazione e l'operazione ad esso associata.

`variable` indicherà la variabile utilizzata per questa ricezione.

`createInstance` indica invece se, ricevuto un messaggio, dovrà essere creata una istanza di questo processo per la gestione di tutta l'interazione. Da notare che questo attributo non può essere associato all'evento di ricezione di un messaggio all'interno del gestore degli eventi, purché il gestore è associato proprio ad una istanza, quindi deve essere già stata creata per avere il gestore stesso.

E' possibile associare ad una *attività* **receive** un *insieme di correlazione* (l'attributo **set** indica il nome dell'insieme a cui si fa riferimento) e indicare se, alla ricezione del messaggio, bisogna inizializzare (**initiate="yes"**) l'insieme di correlazione con le proprietà del messaggio definite dall'insieme stesso.

Non è definita la semantica di un processo in cui due **receive** sono contemporaneamente in esecuzione sullo stesso *partnerLink*, *portType*, *insieme di correlazione* e operazione.

Il costrutto **reply** permette al processo di inviare una risposta al mittente di un messaggio.

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
variable="ncname"? faultName="qname"?
standard-attributes>
standard-elements
<correlations>?
<correlation set="ncname" initiate="yes|no"?>+
</correlations>
</reply>
```

I campi di una *attività* **reply** corrispondono a quelli dell'attività **receive** e il loro significato è lo stesso. Tuttavia l'attività **reply** ha due potenziali forme: se la risposta alla richiesta è normale, il valore del campo **faultName** sarà trascurato e si userà la variabile definita nel campo **variable** come dato del messaggio di risposta mentre se la risposta deve avvisare di un errore riscontrato, si usa l'attributo **faultName** mentre l'attributo **variable**, se definito, indicherà una variabile del tipo di messaggio del corrispondente errore.

2.7.2 Invoke

L'*attività* **invoke** permette ad un processo di invocare un servizio esterno con una iterazione unidirezionale o di richiesta-risposta.

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
```

```
inputVariable="ncname"? outputVariable="ncname"?
standard-attributes>
standard-elements
<correlations>?
<correlation set="ncname" initiate="yes|no"?
pattern="in|out|out-in"/>+
</correlations>
<catch faultName="qname" faultVariable="ncname"?>*
activity
</catch>
<catchAll>?
activity
</catchAll>
<compensationHandler>?
activity
</compensationHandler>
</invoke>
```

Una **invoke** deve essere associata ad una interfaccia esterna. Questa viene definita attraverso gli attributi **partnerLink**, **portType** e **operation** che definiscono il ruolo del servizio in questa interazione, il *portType* WSDL utilizzato per la comunicazione e l'operazione ad esso associata.

Ad una **invoke** sono associate due variabili: una di ricezione (*input*) e una di trasmissione (*output*). Siccome gli *insiemi di correlazione* sono direttamente collegati alle variabili dei messaggi, nel caso di una invocazione sincrona di tipo domanda-risposta si usa l'attributo **pattern** per specificare se la *correlazione* si applica al messaggio di ricezione (**in**) o di trasmissione (**out**) o in entrambi (**out-in**).

Infine ad una *attività* **invoke** possono essere associati sia un gestore di errori (definendone **catch** e **catchAll** come nel caso dell'elemento **faultHandlers**, vedi a tal proposito il paragrafo *2.5.5*) che un gestore di *compensazione* (definito come il **compensationHandler** globale, vedi sempre il paragrafo *2.5.5*). Per quanto riguarda l'utilizzo di questi gestori, è come se l'*attività* fosse racchiusa all'interno di una **scope** definita con tali gestori, il cui **name** è lo stesso della **invoke**. Si rimanda al paragrafo *2.8.6* per maggiori informazioni.

2.7.3 Assign

L'*attività* **assign** può essere usata per definire un arbitrario numero di copie di valori tra coppie di variabili, così come per costruire e inserire nuovi dati tramite espressioni o per copiare un riferimento a un *endpoint*. L'uso di espressioni è motivato dalla necessità di compiere semplici computazioni all'interno del processo, come ad esempio l'incremento di variabili.

```
<assign standard-attributes>
standard-elements
<copy>+
from-spec
to-spec
</copy>
</assign>
```

L'*attività* **assign** copia un valore di tipo compatibile dalla sorgente (***from-spec***) alla destinazione (***to-spec***). La specifica della sorgente deve essere di una dei seguenti tipi:

```
<from variable="ncname" part="ncname"?/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
```

mentre la specifica della destinazione deve essere di una dei seguenti tipi:

```
<to variable="ncname" part="ncname"?/>
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>
```

Nella prima variante l'attributo **variable** fornisce il nome di una variabile. Se il tipo di variabile è un tipo di messaggio WSDL, l'attributo opzionale **part** può essere usato

per fornire il nome della parte interna alla variabile (si rimanda al paragrafo *2.3* per i dettagli sui messaggi WSDL).

Nella seconda variante si permette la manipolazione dinamica dei riferimenti agli *endpoint* associati ad un *partnerLink.* Il valore dell'attributo **`partnerLink`** è il nome di un *partnerLink* definito per il processo. Nel caso del valore sorgente si deve specificare anche il ruolo perché un processo può aver bisogno di comunicare un riferimento ad un *endpoint* corrispondente sia al suo ruolo che a quello del partner. Per il valore di destinazione l'assegnamento è possibile soltanto al ruolo del partner, quindi non c'è bisogno di specificare il ruolo.

Nella terza variante si permette una manipolazione esplicita delle proprietà del messaggio presenti nelle variabili. Questa variante è molto utile nelle definizioni di processi astratti, perchè fornisce un metodo per definire chiaramente come elementi di dati distinti vengano usati nei messaggi.

La quarta variante per la specifica della sorgente permette ai processi di compiere semplici computazioni su proprietà e variabili.

L'ultima variante per la specifica della sorgente permette ad un valore preciso di essere assegnato a una destinazione. Il tipo del valore deve essere dello stesso tipo della destinazione, e può essere indicato usando il meccanismo dei tipi dell'XML Schema.

2.7.4 Throw

L'*attività* **`throw`** può essere usata all'interno di un processo quando si ha bisogno di segnalare esplicitamente un fallimento interno. Ogni fallimento deve avere associato un nome unico che deve essere segnalato all'interno dell'*attività* e, inoltre, si può fornire una variabile che contenga ulteriori informazioni sul fallimento. Il gestore dell'errore può utilizzare queste informazioni per l'analisi e la gestione dell'errore stesso o per spedire messaggi di fallimento ad altri servizi.

```
<throw faultName="qname" faultVariable="ncname"? standard-
attributes>
standard-elements
</throw>
```

In BPEL4WS non viene richiesto che il nome del fallimento sia definito prima del suo uso in una attività **`throw`**. Un nome di fallimento può essere usato direttamente tramite l'utilizzo di un nome appropriato come valore dell'attributo **`faultName`** e fornendo una variabile con i dati dell'errore se richiesto. Questo fornisce un meccanismo molto semplice per introdurre fallimenti specifici di una applicazione.

Un semplice esempio di una *attività* **`throw`** che non fornisce una variabile con informazioni sull'errore è la seguente:

```
<throw xmlns:FLT="http://example.com/faults"
faultName="FLT:OutOfStock"/>
```

2.7.5 Compensate

L'*attività* **`compensate`** è usata per invocare la *compensazione* su uno **`scope`** interno che ha già completato con successo la propria esecuzione.

```
<compensate scope="ncname"? standard-attributes>
standard-elements
</compensate>
```

Questa *attività* costituisce il sostegno del framework di gestione degli errori controllato dalle applicazioni di BPEL4WS, e può essere usata solo nelle seguenti parti di un processo di business:

- in un gestore degli errori dello **`scope`** esterno che racchiude lo **`scope`** per il quale si deve gestire la *compensazione*;

- nel gestore della *compensazione* dello **`scope`** esterno che racchiude lo **`scope`** per il quale si deve gestire la *compensazione*.

Se uno **`scope`** che deve essere compensato poiché il suo nome coincide con

l'attributo **scope** della **compensate** è annidato in un ciclo, le istanze del gestore nelle iterazioni successive sono invocate in ordine inverso.

Se il gestore per uno **scope** è assente, il gestore di default invoca i gestori della *compensazione* per gli **scope** interni a quello riferito nell'ordine inverso di completamento di questi **scope**. Questo comportamento è invocato esplicitamente nel caso in cui sia omesso il nome dello **scope** nell'*attività* **compensate**. Questo è utile quando un gestore di errori o *compensazione* padre deve effettuare altri compiti, come aggiornare variabili o inviare notifiche all'esterno, in aggiunta a invocare la *compensazione* di default per gli **scope** figli.

Da notare che l'attività <**compensate/>** in un gestore associato allo **scope** S causa l'invocazione nell'ordine di default dei gestori di *compensazione* per gli scope completati direttamente annidati in S. L'uso di questa *attività* può essere mescolata con ogni altro comportamento specificato dall'utente ad eccezione dell'invocazione esplicita di <**compensate scope="Sx"/>** per lo **scope** Sx annidato direttamente in S. L'invocazione esplicita di *compensazione* per tale scopo annidato in S disabilita la disponibilità di compensazione con ordine di default, come ci si aspetta.

2.7.6 Terminate e la terminazione delle attività

L'*attività* **terminate** può essere usata per terminare immediatamente il comportamento di una istanza di processo di business all'interno del quale è stata eseguita l'*attività*.

```
<terminate standard-attributes>
standard-elements
</terminate>
```

In genere, in presenza di una terminazione, sia implicita (a causa di errori) che esplicita (generata dall'*attività* **terminate**), tutte le *attività* correntemente in esecuzione devono essere terminate non appena possibile senza gestione di errori o comportamenti di *compensazione*. Una **assign** in esecuzione, così come una valutazione di espressione, viene considerata di così breve durata di vita che si permette il completamento invece

che imporne la terminazione. La nozione di terminazione non si applica ad *attività* quali **`empty`**, **`terminate`** e **`throw`**. Tutte le altre *attività semplici* terminano. Quelle *strutturate* fanno terminare le loro attività interne e terminano esse stesse, ad eccezione della **`scope`** che ha un comportamento diverso. Questa attività infatti, anzi che venir terminata, genera un errore **`bpws:forcedTermination`**, che viene gestito, qualora il gestore sia ancora installato (e quindi non sia stata ancora gestita nessuna eccezione per quello **`scope`**).

Tuttavia questo gestore è particolare perché non propaga tale errore allo **`scope`** superiore, qualora non sia stato definito, e questo perché lo **`scope`** padre è già vittima di un errore di terminazione forzata, quindi non avrebbe senso propagare un altro errore.

Se lo **`scope`** stava già gestendo un errore si permette la terminazione di questa gestione.

Da notare che la terminazione forzata di **`scope`** annidati avviene a partire dal più interno come risultato della regola che il comportamento di gestione di errori inizia con la terminazione implicita e ricorsiva di tutte le *attività* attive racchiuse direttamente all'interno del suo **`scope`** associato.

L'*attività* **`terminate`** è disponibile solo nei processi eseguibili.

2.7.7 Wait

L'*attività* **`wait`** permette di attendere per un determinato periodo di tempo o fino a che una certo periodo sia passato (deve essere scelto esattamente uno dei criteri di scadenza).

```
<wait (for="duration-expr" | until="deadline-expr") standard-
attributes>
standard-elements
</wait>
```

Un tipico uso di questa attività è quello di attendere un certo momento per invocare una operazione.

2.7.8 Empty

C'è spesso bisogno di non compiere nulla, per esempio quando è necessario catturare un errore e sopprimerlo o per sincronizzare attività concorrenti. L'*attività* **`empty`** è usata per questo scopo.

```
<empty standard-attributes>
standard-elements
</empty>
```

2.8 Attività strutturate

Le *attività strutturate*, come anticipato, prescrivono l'ordine di esecuzione delle attività di cui sono composte. Descrivono come un processo di business sia creato a partire dalle *attività* di base che questo esegue composte in strutture che ne esprimono il modello di controllo, il flusso dei dati, la gestione degli errori e di eventi esterni e la coordinazione di scambio di messaggi tra istanze di processo implicate in un protocollo di business.

Le *attività strutturate* in BPEL4WS includono:

- controllo sequenziale ordinario tra attività fornito da **`sequence`**, **`switch`** e **`while`**;
- concorrenza e sincronizzazione tra attività fornito da **`flow`**;
- scelta non deterministica basata su eventi esterni fornito da **`pick`**;

tali *attività* possono essere usate ricorsivamente. Un punto chiave da capire è che queste *attività strutturate* possono essere annidate e combinate in maniera arbitraria. Come è stato già detto il termine *attività* verrà usato quindi ad indicare una qualsiasi *attività*, sia essa semplice o strutturata.

2.8.1 Sequence

L'*attività* **sequence** contiene una o più *attività* che vengono eseguite in maniera sequenziale, nell'ordine in cui sono elencate all'interno dell'elemento **sequence**, cioè in ordine lessicale. L'*attività* è completata quando l'*attività* finale della sequenza è stata completata.

```
<sequence standard-attributes>
standard-elements
activity+
</sequence>
```

2.8.2 Flow

L'*attività* **flow** fornisce concorrenza e sincronizzazione.

```
<flow standard-attributes>
standard-elements
<links>?
<link name="ncname">+
</links>
activity+
</flow>
```

Gli elementi e gli attributi standard per le attività annidati in una **flow** sono particolarmente significanti perché tali caratteristiche standard esistono principalmente per fornire alle *attività* semantica relativa al **flow** stesso.

L'effetto semantico più rilevante che si ottiene raggruppando un insieme di attività in una **flow** è di permettere la concorrenza. La **flow** è completata quando tutte le *attività* al suo interno sono state completate, e in questo caso per una *attività* essere completata include la possibilità di essere stata saltata se la sua condizione di attivazione risulta essere falsa (*Dead-Path-Elimination*).

Quindi una *attività* di **flow** manda in esecuzione in maniera concorrente l'insieme di *attività* direttamente annidate al suo interno. In più permette espressioni di dipendenza

di sincronizzazione tra *attività* che sono annidate direttamente o indirettamente al suo interno. Queste dipendenze vengono specificate attraverso il costrutto **`link`**, e tutti i nomi dei **`link`** devono essere specificati separatamente all'interno della **`flow`**. Per stabilire una dipendenza tra le *attività* A e B (ovvero un **`link`** tra A e B) si usano due elementi: in A verrà definito il **`source`** mentre in B il **`target`**. Per il funzionamento dei **`link`** si rimanda alla trattazione nel paragrafo *2.6.*

2.8.3 Switch

L'*attività* **`switch`** permette un comportamento condizionale. L'*attività* consiste di una lista di uno o più rami condizionali definiti da elementi **`case`**, seguiti dal ramo opzionale **`otherwise`**.

```
<switch standard-attributes>
standard-elements
<case condition="bool-expr">+
activity
</case>
<otherwise>?
activity
</otherwise>
</switch>
```

I rami **`case`** sono considerati in ordine in cui appaiono. Viene scelto il primo ramo la cui condizione viene verificata e l'*attività* correlata a questo ramo viene eseguita. Se nessun ramo **`case`** può essere scelto si esegue l'*attività* del ramo **`otherwise`**, che può essere definito esplicitamente o implicitamente considerato presente e formato da una *attività* **`empty`**.

L'*attività* **`switch`** è completata quando l'*attività* del ramo selezionato è stata completata.

2.8.4 While

L'*attività* **`while`** permette l'esecuzione ripetuta di una specifica *attività*. Tale ripetizione si attua finché una condizione specificata non risulta essere falsa.

```
<while condition="bool-expr" standard-attributes>
standard-elements
activity
</while>
```

Ricordiamo che, in caso di terminazione forzata, l'iterazione sarebbe interrotta e la terminazione sarebbe propagata all'*attività* racchiusa nella **`while`**. Inoltre, per quanto riguarda i *link*, quelli che interessano l'*attività* interna non possono superare i confini della **`while`** stessa.

2.8.5 Pick

L'*attività* **`pick`** aspetta l'occorrenza di un insieme di eventi ed esegue l'*attività* collegata a quello che si manifesta. Generalmente gli eventi sono mutuamente esclusivi. Qualora si generassero due eventi verrà comunque eseguito solo il primo, a meno che i due eventi non si generino quasi simultaneamente, nel qual caso la scelta sarà legata all'implementazione, oltre che a fattori legati al tempo.

```
<pick createInstance="yes|no"? standard-attributes>
standard-elements
<onMessage partnerLink="ncname" portType="qname"
operation="ncname" variable="ncname"?>+
<correlations>?
<correlation set="ncname" initiate="yes|no"?>+
</correlations>
activity
</onMessage>
<onAlarm (for="duration-expr" | until="deadline-expr")>*
activity
</onAlarm>
```

```
</pick>
```

La **pick** è formata da un insieme di rami del tipo evento-*attività*, ed esattamente uno di questi verrà scelto, in base all'occorrenza dell'evento associato che si manifesterà prima degli altri.

Dopo che la **pick** avrà accettato un evento nessun altro evento potrà essere considerato da questa stessa *attività*. Gli eventi considerati riguardano l'arrivo di messaggi sotto forma di invocazioni ricevute di tipo unidirezionale o richiesta-risposta, o un "allarme" basato su un timer.

Una forma speciale di **pick** è usata quando potrebbe essere creata una istanza del processo di business come risultato della ricezione di uno di un insieme di possibili messaggi. In questo caso la **pick** ha l'attributo **createInstance** definito come **yes** (il valore di default per questo attributo è **no**). In questo caso tutti gli eventi della **pick** devono essere messaggi di ricezione ed ognuno di essi è equivalente ad una **receive** con l'attributo **createInstance=yes** (vedere a riguardo la **receive** al paragrafo *2.7.1*). Non è permesso nessun allarme in questo caso speciale.

Ogni *attività* **pick** deve includere almeno un evento **onMessage** (ricezione di messaggio). La semantica di un evento **onMessage** è identica a una attività **receive** ad eccezione della natura opzionale dell'attributo **variable** e il vincolo che riguarda l'attivazione simultanea di azioni di ricezione in conflitto. Per questo secondo caso si ricordi che la semantica di un processo in cui due o più azioni di ricezione sono definite sugli stessi *partnerLink*, *portType*, operazioni e *insiemi di correlazione* e vengono eseguite contemporaneamente non è definita. Per questo vincolo l'attivazione di una **onMessage** equivale all'attivazione della corrispondente attività **receive**.

La **pick** è completa quando è stato selezionato uno dei rami e l'*attività* associata ha terminato la sua esecuzione.

2.8.6 Scope

Il contesto per il comportamento di ogni attività è fornito da uno **scope**, che definisce gestori di errori, eventi, *compensazione* oltre che variabili e *insiemi di correlazione*.

Tutti gli elementi dello **scope** sono sintatticamente opzionali e alcuni hanno una

semantica di default quando sono omessi.

```
<scope variableAccessSerializable="yes|no" standard-attributes>
standard-elements
<variables>?
... vedi paragrafo 2.5.3 per la sintassi ...
</variables>
<correlationSets>?
... vedi paragrafo 2.5.4 per la sintassi ...
</correlationSets>
<faultHandlers>?
... vedi paragrafo 2.5.5 per la sintassi ...
</faultHandlers>
<compensationHandler>?
... vedi paragrafo 2.5.5 per la sintassi ...
</compensationHandler>
<eventHandlers>?
... vedi paragrafo 2.5.5 per la sintassi ...
</eventHandlers>
activity
</scope>
```

Ogni **scope** ha una *attività* principale che definisce il suo comportamento normale, e può essere una *attività strutturata* molto complessa, con un livello di annidamento arbitrario. Lo **scope** è condiviso da tutte le *attività* annidate, che quindi utilizzeranno variabili, gestori e *insiemi di correlazione* gestiti al suo interno.

Le variabili definite all'interno di uno **scope** sono visibili e utilizzabili solo nello **scope** stesso, e non esteriormente. Anche i gestori (e ciò è molto importante per la *compensazione*) utilizzano le variabili interne, quindi uno **scope** può essere usato per salvare valori durante l'esecuzione di un processo e riutilizzarli in caso di *compensazione*.

Un gestore di *compensazione* è installato se lo **scope** ha terminato la sua esecuzione normalmente. Se in uno **scope** S viene lanciato un errore, anche se questo viene gestito

il gestore della *compensazione* non viene installato. Tuttavia lo **`scope`** padre di quello che ha gestito l'errore continua la sua esecuzione, e i **`link`** del gestore degli errori di S vengono valutati.

Se non viene dichiarato un gestore di *compensazione* per uno **`scope`**, quello di default esegue tutti i gestori di *compensazione* disponibili negli **`scope`** che racchiude nell'ordine inverso di completamento.

Un gestore degli errori viene installato quando lo **`scope`** inizia la sua esecuzione e disattivato quando la termina. Questo significa che non possono essere gestiti due errori nello stesso **`scope`**. La gestione degli errori comincia con la terminazione di tutte le attività (si veda a tal proposito il paragrafo *2.7.6*). Un errore che non viene gestito da uno **`scope`** viene trasmesso (e quindi lanciato) all'interno dello **`scope`** padre.

Se viene omesso un gestore degli errori, quello di default ha il seguente comportamento: esegue tutti i gestori di *compensazione* come per il gestore di *compensazione* di default, dopodiché lancia lo stesso errore nello **`scope`** padre.

Per quanto riguarda il gestore degli eventi, questo è installato quando l'*attività* interna dello **`scope`** viene messa in esecuzione e disattivato quando lo **`scope`** termina. Se un evento è in fase di gestione, se ne permette sempre il completamento.

Quando l'attributo **`variableAccessSerializable`** ha come valore **`"yes"`**, viene fornito un controllo di concorrenza nel governare l'accesso alle variabili condivise. Questo **`scope`** viene definito un *serializable scope* e non può essere annidato (deve cioè essere una foglia dell'albero delle *attività*). Se due *serializable scope* S1 e S2 accedono a un insieme comune di variabili esterne a loro per operazioni di scrittura o lettura, la semantica della serializzabilità assicura che il risultato del loro comportamento non sarà differente se tutte le attività in conflitto (lettura-scrittura e scrittura-scrittura) su ogni variabile condivisa sono concettualmente riordinate in modo tale sia che S1 completi tutte le sue *attività* prima di S2 sia vice versa. Il meccanismo per garantire ciò è dipendente dall'implementazione.

L'uso dei gestori di errore è governato dalle seguenti regole: la serializzabilità è mantenuta anche all'interno del gestore, che quindi manterrà l'accesso alle variabili condivise fino al completamento, il gestore della *compensazione* non mantiene la serializzabilità ma mantiene una immagine dello stato di esecuzione.

E' utile notare che la semantica dei *serializable scope* è molto simile al livello di isolazione "serializzabile" standard usato nelle transizioni dei database.

Capitolo 3 - YAWL (Yet Another Workflow Language)

3.1 Introduzione

I linguaggi di specifica per i workflow sono innumerevoli e differenti tra loro per varie caratteristiche, questo principalmente perché non si è mai raggiunto un accordo sui fondamenti formali di tali linguaggi, non ci sono mai state standardizzazioni nei confronti di questi strumenti e delle funzionalità che devono garantire, così come non esiste un metodo per confrontare la loro potenza di espressione.

L'iniziativa sui pattern per workflow mira a stabilire un approccio più strutturato per specificare le dipendenze dei flussi di controllo nei linguaggi di workflow. Basandosi sull'analisi di applicazioni e di sistemi di gestione di workflow esistenti, questa iniziativa ha identificato una collezione di pattern che corrisponde alla tipica dipendenza di flussi di controllo incontrati nelle specifiche dei workflow, e ha documentato i metodi per catturare queste dipendenze nei linguaggi di workflow. Questi pattern vengono usati come stimatori per confrontare i linguaggi di definizione di processi.

Mentre i pattern per workflow forniscono un approccio pragmatico alla specifica dei flussi di controllo nei workflow, le reti di Petri garantiscono un approccio più teoretico.

Le reti di Petri furono originariamente sviluppate tra gli anni '60 e '70, è furono presto ritenute uno dei migliori linguaggi di descrizione e analisi della sincronizzazione, comunicazione e condivisione di dati tra processi concorrenti. Tuttavia tentare di usare le reti di Petri nella pratica rivelò due seri problemi: primo non esisteva concetto di dato e quindi il modello spesso diventava eccessivamente ampio, perché tutte le manipolazioni di dati dovevano essere rappresentate all'interno della struttura; secondo non c'erano concetti gerarchici, e quindi non era possibile costruire un modello grande a partire da un insieme di modelli più piccoli costruiti

separatamente che avessero interfacce ben definite.

Lo sviluppo delle reti di Petri di alto livello nei tardi anni ’70 e delle reti di Petri gerarchiche nei tardi anni ’80 rimossero questi due seri problemi. Le reti di Petri Colorate (*Coloured Petri Nets*, chiamate anche *CP-nets* oppure *CPN*) sono uno dei dialetti più conosciuti delle reti di Petri di alto livello, e incorporano sia la strutturazione dei dati che la decomposizione gerarchica senza compromettere la qualità delle reti di Petri originali.

Nonostante l’enorme potere espressivo, dimostrato anche dall’elevato numero di pattern per workflow supportati, le reti di Petri, anche nelle varianti ad alto livello, non permettono di gestire tutti i tipi di pattern individuati. Tra quelli non supportati ci sono quelli di cancellazione (che consistono nella possibilità di cancellare l’esecuzione di uno o di un gruppo di *task*, che sono le attività di un workflow), il pattern di unione sincronizzata (dove tutti i thread attivi necessitano di essere uniti, e i rami che non possono diventare attivi devono essere ignorati), e i pattern che considerano istanze multiple attive della stessa attività nella stessa condizione.

Queste carenze hanno portato allo sviluppo di YAWL il quale combina l’esperienza ottenuta dai pattern per workflow con i benefici delle reti di Petri. E’ però da notare che YAWL non è costruito sopra le reti di Petri, anche se ne è ispirato, e la sua semantica non ne fa riferimento, ma è definita in termini di sistema di transizione.

YAWL è quindi un linguaggio molto espressivo, nel senso che dà supporto diretto per tutti i principali pattern per workflow, mentre gli altri linguaggi danno supporto solo per un ristretto sottoinsieme di questi. In più YAWL ha una semantica formale e offre una rappresentazione grafica di molti dei suoi concetti. Questo permette di usarlo come linguaggio intermedio per avere il supporto nella traduzione di workflow specificati in linguaggi differenti.

3.2 Pattern di riferimento

Un pattern viene definito come una astrazione da una forma concreta che ricorre in contesti specifici, ovvero è un modello che spesso ricorre in vari workflow con delle

piccole differenze. I pattern per la valutazione dei vari linguaggi di workflow sono 20. Verranno elencati di seguito raggruppati in 6 categorie:

Categoria 1: Pattern di base per i flussi di controllo.

Questi sono costrutti base presenti in molti linguaggi di workflow per modellare l'istradamento sequenziale, parallelo e condizionale.

1. **Sequenza** (*Sequence*) – un *task* nel processo workflow è attivato dopo il completamento di un altro task nello stesso processo.
2. **Split parallelo** (*Parallel Split*) – un punto nel processo workflow dove un singolo thread di controllo si divide in multipli thread di controllo che possono essere eseguiti in parallelo, cioè permettendo ai *task* di essere eseguiti in maniera simultanea o in qualsiasi ordine
3. **Sincronizzazione** (*Synchronization*) – un punto nel processo workflow dove sottoprocessi o *task* multipli convergono in un singolo thread di controllo, ottenendo quindi la sincronizzazione di multipli thread. E' assunzione di questo pattern che ogni ramo in ingresso di un sincronizzatore viene eseguito solo una volta (se non è il caso in esame, riferirsi ai pattern 13 e 15)
4. **Scelta esclusiva** (*Exclusive Choice*) – un punto nel processo workflow dove, basandosi su una decisione o su dati di controllo del workflow, viene scelto uno di diversi rami.
5. **Merge semplice** (*Simple Merge*) – un punto nel processo workflow dove due o più rami alternativi si uniscono senza sincronizzazione. E' assunzione di questo pattern che nessun ramo alternativo è mai eseguito in parallelo (altrimenti si veda il pattern 8 o 9).

Categoria 2: Pattern di ramificazione avanzata e sincronizzazione.

Questi pattern trascendono quelli base per permettere tipi avanzati di comportamenti nella divisione (*split*) e nell'unione (*join*). Un esempio è il pattern 7 che si comporta come un join AND o XOR in base al contesto.

6. **Scelta multipla** (*Multiple Choice*) – un punto nel processo workflow dove, in base a una decisione o ai dati di controllo del workflow, vengono scelti un certo numero di rami.
7. **Merge di sincronizzazione** (*Synchronizing Merge*) – Un punto nel processo di workflow dove cammini multipli convergono in un singolo thread. Se più di un cammino è attivo, è necessario che ci sia una sincronizzazione dei thread attivi. Se invece è attivo solo un cammino, i rami alternativi possono riconvergere senza sincronizzazione. E' assunzione di questo pattern che un ramo che è già stato attivato non può essere riattivato mentre l'unione sta ancora aspettando che altri rami siano completi.
8. **Merge multiplo** (*Multi-Merge*) – un punto nel processo workflow dove due o più rami riconvergono senza sincronizzazione. Se è stato attivato più di un ramo, possibilmente in maniera concorrente, il *task* che viene dopo l'unione dei rami viene attivato una volta per ogni ramo che raggiunge il punto di unione.
9. **Discriminator** – un punto nel processo workflow che aspetta che un ramo in ingresso sia completo prima di attivare il *task* seguente. Da quel momento in poi aspetta che gli altri rami siano completi e li ignora. Quando tutti i rami sono stati valutati si rinizializza in modo che possa essere riutilizzato da capo (questo è importante soprattutto per i cicli).

Categoria 3: Pattern strutturali.

Nei linguaggi di programmazione è abbastanza naturale trovare un blocco strutturato in cui è facilmente identificabile un punto di ingresso e uno di uscita. Nei linguaggi grafici in cui è permesso il parallelismo questo requisito è spesso considerato troppo restrittivo. Comunque sono stati identificati pattern che permettono una struttura meno rigida.

10. **Cicli arbitrari** (*Arbitrary Cycles*) – un punto nel processo workflow dove uno o più *task* possono essere eseguiti ripetutamente.
11. **Terminazione implicita** (*Implicit Termination*) – dato un sottoprocesso, questo dovrebbe terminare quando non ha più niente da fare, ovvero quando non ci sono

più *task* attivi nel workflow e nessun altro *task* può essere attivato (e allo stesso tempo il workflow non è in stallo).

Categoria 4: Pattern che interessano istanze multiple.

Nel contesto di un singolo caso (cioè di una singola istanza di workflow) qualche volta alcune parti del processo devono essere istanziate più volte.

12. **Istanze multiple senza sincronizzazione** (*Multiple Instances Without Synchronization*) – all'interno del contesto di una singola istanza di workflow possono essere create istanze multiple di un *task*, ovvero c'è la possibilità di creare nuovi thread indipendenti tra loro, e inoltre non c'è necessità di sincronizzare questi thread.
13. **Istanze multiple con una conoscenza a priori durante la progettazione** (*Multiple Instances With A Priori Design Time Knowledge*) – per una istanza di processo un *task* viene attivato più volte. Il numero di istanze di un dato *task* per una data istanza di processo è conosciuta a tempo di progettazione. Una volta che tutte le istanze sono complete devono essere eseguiti altri *task*.
14. **Istanze multiple con una conoscenza a priori a tempo di esecuzione** (*Multiple Instances With A Priori Runtime Knowledge*) – per una istanza di processo un *task* viene attivato più volte. Il numero delle istanze di un dato *task* per una data istanza di processo varia e può dipendere da caratteristiche del processo o dalla disponibilità di risorse, ma è mota in un certo punto dell'esecuzione, prima che le istanze del *task* vengano create. Una volta che tutte le istanze sono complete una qualche altra attività deve essere abilitata.
15. **Istanze multiple senza una conoscenza a priori a tempo di esecuzione** (*Multiple Instances Without A Priori Runtime Knowledge*) – per una istanza di processo un task viene abilitato più volte. Il numero di istanze di un dato *task* per una data istanza di processo non è nota a tempo di progettazione né in nessun punto dell'esecuzione, prima che le istanze del *task* debbano essere create. Una volta che tutte le istanze sono complete un qualche altro *task* viene eseguito. La differenza col pattern 14 è che possono esserne create nuove istanze anche mentre alcune sono in esecuzione o già complete.

Categoria 5: Pattern basati sullo stato.

I sistemi tipici di workflow si focalizzano solo su attività ed eventi e non sugli stati. Ciò limita l'espressività dei linguaggi di workflow perché non è possibile avere pattern dipendenti da stati come il pattern 18 (*Milestone*).

16. **Scelta differita** (*deferred choice*) – un punto nel processo workflow dove viene scelto uno di più rami. In contrasto con lo split XOR, la scelta non è fatta esplicitamente (ovvero non è basata su dati o decisioni) ma sono possibili più alternative. Tuttavia, in contrasto con lo split AND, solo una delle alternative viene eseguita. Questo significa che una volta che viene attivato un ramo gli altri sono disattivati. E' importante notare che la scelta è ritardata finché il processo non attiva uno dei rami, ovvero la scelta viene fatta il più tardi possibile.
17. **Interleaved Parallel Routing** – un insieme di *task* è eseguito in un ordine arbitrario: ogni *task* nell'insieme è eseguito, l'ordine viene deciso a tempo di esecuzione e non vengono mai eseguiti due *task* nello stesso momento (ovvero non ci sono mai due *task* dell'insieme attivi contemporaneamente per la stessa istanza di workflow)
18. **Milestone** – l'attivazione di un *task* dipende dal caso in cui si raggiunga uno stato specifico, ovvero il *task* è attivato se un certo punto cardine (*milestone*, ovvero la pietra migliare) viene raggiunto quando non è ancora espirato il suo tempo di vita. Considerati tre *task* A, B e C, supponiamo che A è attivato solo se B è stato eseguito e C no, ovvero A non è attivato prima dell'esecuzione di B né dopo l'esecuzione di C. Lo stato tra B e C è modellato dalla posizione *m*, che è la milestone per A.

Categoria 6: Pattern di cancellazione.

L'occorrenza di un evento (come, ad esempio, un cliente che cancella un ordine) può portare alla cancellazione di uno o più *task*. In alcuni casi tali eventi possono portare all'annullamento dell'intera istanza di processo.

19. **Cancellazione di un *task*** (*Cancel Activity*) – un *task* in esecuzione viene disabilitato, ovvero un thread in attesa dell'esecuzione di una attività viene rimosso.
20. **Cancellazione di un processo** (*Cancel Case*) – una istanza di processo viene completamente rimossa (anche se alcune parti di un processo sono istanziate più volte, tutti i discendenti sono rimossi).

3.3 Il linguaggio YAWL

In questo paragrafo verrà introdotto il linguaggio YAWL e la notazione grafica ad esso associata. Come detto precedentemente, YAWL è ispirato alle reti di Petri, ma, per evitarne le limitazioni, è stato esteso con funzionalità per facilitare i pattern che utilizzano istanze multiple, pattern di sincronizzazione avanzata e pattern di cancellazione, oltre ad avere costrutti per la decomposizione gerarchica e per la gestione di dati arbitrariamente complessi.

Per ottenere tali risultati YAWL estende la classe delle reti di workflow esistenti con elementi quali istanze multiple, *task* compositi, OR-join, rimozione di token e transizioni connesse direttamente. Come abbiamo detto YAWL è ispirato alle reti di Petri ma non è solo un pacchetto di macro costruito sopra le reti di Petri ad alto livello: è un linguaggio completamente nuovo con la sua semantica e appositamente costruito per la specifica di workflow.

La seguente figura mostra gli elementi grafici di YAWL.

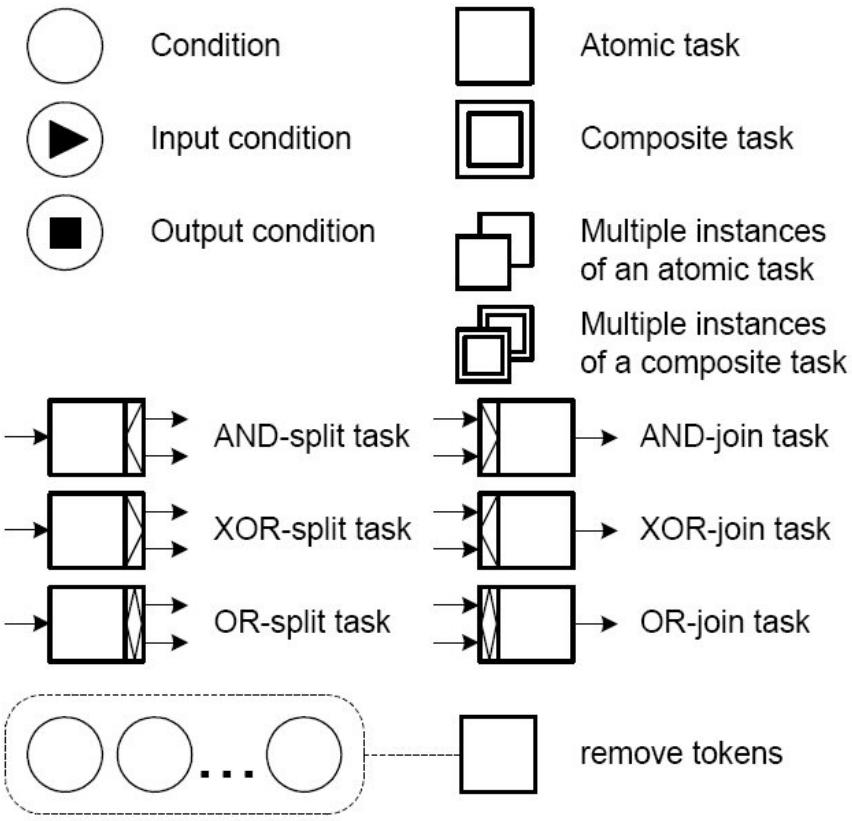


Figura 3.1: rappresentazione grafica dei principali elementi di YAWL

Una specifica di workflow in YAWL è un insieme di definizioni di processo che formano una gerarchia. I task possono essere atomici o compositi. Ogni task composito si riferisce a una definizione di processo a un livello più basso della gerarchia (a cui ci si riferisce come alla sua *decomposizione*). I task atomici formano le foglie della struttura a grafo.

C'è una definizione di processo senza alcun task composito che lo riferisce: questa definizione di processo è chiamata *workflow di livello più alto* (*top level workflow*), oppure *workflow radice* (*root workflow*) e forma la radice della struttura a grafo che rappresenta la gerarchia delle definizioni del processo.

Ogni definizione di processo consiste in *task* (sia compositi che atomici) e

condizioni. Ogni definizione di processo ha una unica *condizione di input* (*Input condition*) e una unica *condizione di output* (*Output condition*), rappresentate come mostrato nella figura *3.1*.

Al contrario di quanto succede nelle reti di Petri è possibile connettere due oggetti di 'transizione', come *task* atomici o composti, direttamente tra di loro senza usare oggetti intermedi (gli equivalenti delle *condizioni*). Per il significato in termini di reti di Petri, questo è equivalente ad avere una condizione nascosta, ovvero una condizione implicita viene aggiunta ad ogni connessione diretta.

Ogni *task*, sia esso atomico o composito, può avere istanze multiple, come indicato graficamente nella figura, di cui è possibile specificare la quantità in termini di limite inferiore e superiore di istanze create all'inizializzazione del *task*. In più è possibile determinare che il *task* termina quando una certa soglia di numero di istanze è stata completata e, raggiunta tale soglia, tutte le istanze ancora in esecuzione vengono terminate e il task termina l'esecuzione. Infine si può decidere che il numero di istanze sia *statico* ovvero deciso all'inizializzazione del *task*, oppure *dinamico*, cioè è permessa l'aggiunta di ulteriori istanze nel mentre che il *task* non è ancora completo. In questo modo vengono supportati tutti i pattern che richiedono o gestiscono istanze multiple e, in aggiunta, il pattern **Discriminator** (Pattern 9), assumendo istanze multiple dello stesso *task*.

Sono adottati gli *split* e i *join* AND/XOR (il *join* indica il comportamento che viene assunto per quanto riguarda il flusso in ingresso al *task*, mentre lo *split* indica il comportamento che viene assunto per quanto riguarda il flusso in uscita) ma in aggiunta si usano anche gli OR che permettono di usare il pattern per le scelte multiple (Pattern 6), in caso di *split*, e quello per l'unione sincronizzata (Pattern 7) in caso di *join*.

Per finire viene introdotta la notazione per rimuovere i token da una locazione, a prescindere da quanti token essa contenga. La notazione usata è visibile nella figura. Il *task* di rimozione non ha una dipendenza di nessun tipo col numero di token nella sua area di cancellazione, solamente alla sua esecuzione tutti i token che si troveranno nell'area di cancellazione verranno rimossi. Questo implementa il pattern di cancellazione.

Per la definizione formale del linguaggio si rimanda a [2]. In questo capitolo verrà data una spiegazione informale della semantica del linguaggio e verranno proposti alcuni esempi tratti da [2].

Intanto bisogna dire che una rete workflow di YAWL è formata da una *condizione di input*, una *condizione di output*, un insieme di *task*, con associato una molteplicità, un insieme di cancellazione e uno *split* e un *join* scelto tra AND, OR e XOR, un insieme di condizioni e la relazione di *flusso* che, a partire dalla *condizione di input* fino alla *condizione di output* collega un elemento (*task* o *condizione* che non sia *di output*) a quello successivo (*task* o *condizione* che non sia *di input*), in modo tale che tutti gli elementi (*task* e *condizioni*) si trovino su un possibile cammino che, partendo dalla *condizione di input*, permette di raggiungere la *condizione di output*.

Una specifica per un workflow YAWL è un insieme di reti workflow tale che sia definito un *workflow di livello più alto*, tutti i nomi dei *task* siano unici e tutti i *task* compositi siano associati alla rete workflow che determina la loro decomposizione.

3.4 Gestione dei dati in un workflow YAWL

In un workflow YAWL esiste la possibilità di specificare i dati che sono utilizzati. L'uso delle variabili può essere quello di passare dati dall'utente del workflow all'engine YAWL oppure quello di scambiare i dati dal workflow ai *servizi web* invocati dall'engine e vice versa.

I dati in un workflow creato con l'editor YAWL vengono gestiti in questo modo: ogni rete può avere un numero arbitrario di variabili. Anche i singoli *task* possono avere delle variabili locali, che servono per la loro esecuzione, ma i valori di input devono provenire da una variabile della rete a cui appartiene e quelli di output devono essere salvati in una variabile della rete (i task definiscono un *mapping* in ingresso e in uscita tra le loro variabili e quelle della rete di cui fan parte). In pratica non è possibile scambiare direttamente valori tra *task* ma bisogna usare le variabili della rete come tramite, come se fossero delle variabili globali.

I tipi di variabile supportati sono boolean, date, double, long, string e time. E' possibile però creare tipi complessi descrivendoli tramite il linguaggio XMLSchema.

3.5 Esempi di workflow YAWL

Nelle prossime figure verranno visualizzati i workflow di tre esempi tratti da [2] che riguardano la gestione di dichiarazioni multiple di testimoni (e quindi la gestione di istanze multiple di *task*). La prima specifica di workflow manda in esecuzione da 1 a 10 istanze del *task* **process_witness_statements** (*dichiarazioni dei testimoni del processo*), dopo aver completato il *task* iniziale di registrazione dei testimoni (**register_witnesses**). Quando tutte le istanze sono terminate, viene eseguito il *task* di archiviazione (**archive**).

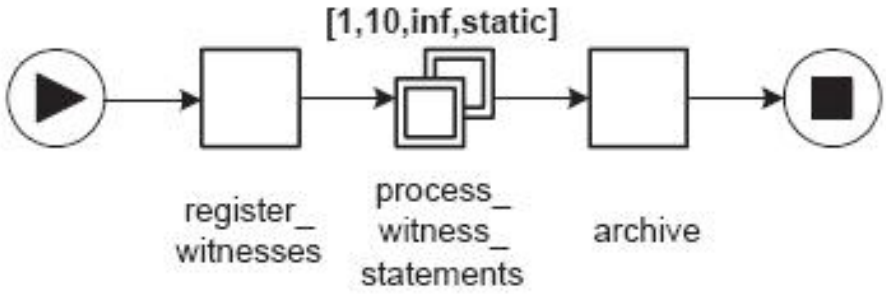


Figura 3.2: primo esempio di gestione di istanze multiple

Nel workflow seguente invece vengono eseguite un numero arbitrario di istanze per il *task composito* e si permette la creazione di nuove istanze.

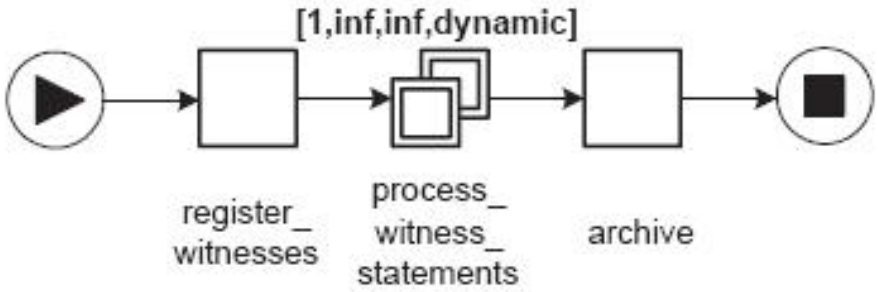


Figura 3.3: secondo esempio di gestione di istanze multiple

L'ultimo workflow, mostrato nella figura seguente, esegue da 1 a 10 istanze del

task composito, ma termina se almeno tre di esse terminano.

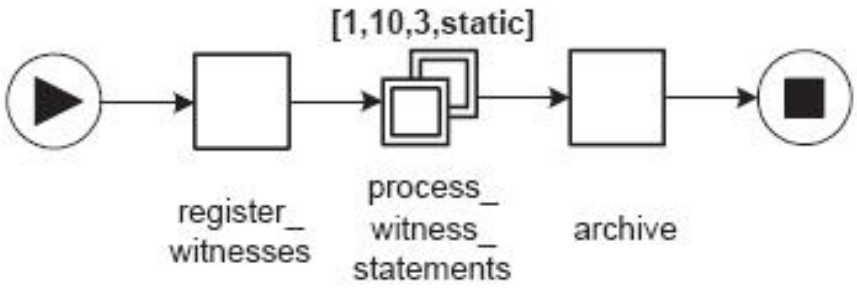


Figura 3.4: terzo esempio di gestione di istanze multiple

Questi tre esempi dimostrano che YAWL garantisce un supporto diretto per i pattern che gestiscono le istanze multiple.

I prossimi esempi chiarificano il significato dello *split* e del *join* di tipo OR. Per questo caso si prende in esame un sistema di prenotazioni di viaggi, in cui un utente può scegliere se prenotare il volo, l'hotel e la macchina in una qualsiasi combinazione (solo volo, solo hotel, solo macchina, volo e hotel, volo e macchina, hotel e macchina oppure tutti e tre). Dopo la prenotazione sarà presente un *task* di pagamento.

Nel primo esempio, graficamente rappresentato nella seguente figura, il *task* di registrazione manda in esecuzione uno o più dei tre *task* che indicano il tipo di prenotazione. Per ogni tipo di prenotazione verrà eseguito il *task* di pagamento.

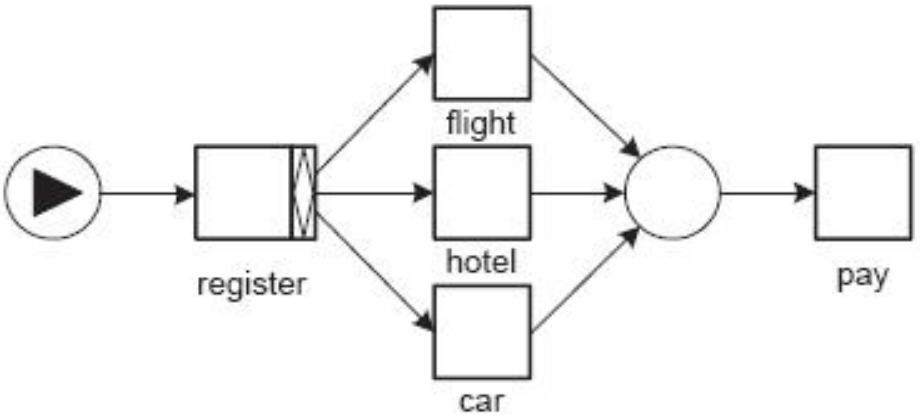


Figura 3.5: funzionamento dell'OR-split

Il seguente esempio invece limita l'esecuzione del pagamento ad una volta sola, quando tutti i *task* in esecuzione (che possono essere in numero da uno a tre) sono stati completati.

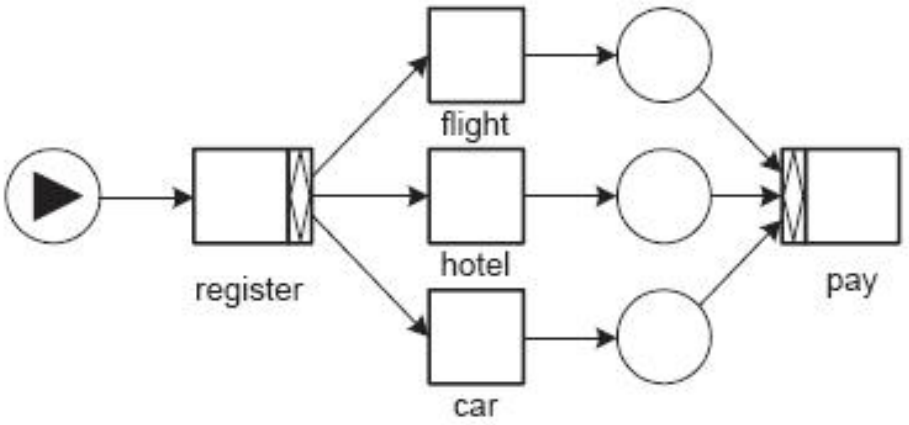


Figura 3.6: funzionamento dell'OR-join

L'esempio che segue invece non ha molto senso dal punto di vista del significato di prenotazione che dovrebbe avere il workflow, ma viene riportato perché implementa il pattern **Discriminator** (Pattern 9).

In questo esempio dopo il *task* di registrazione vengono eseguiti tutti e tre (notare lo *split* di tipo AND) i *task* seguenti, ma il primo che termina farà eseguire l'attività di

pagamento, che però cancellerà anche tutti i token rimasti in esecuzione. Questo significa che il pagamento verrà compiuto solo una volta, quando il primo dei tre *task* lanciati dopo la registrazione sarà stato terminato.

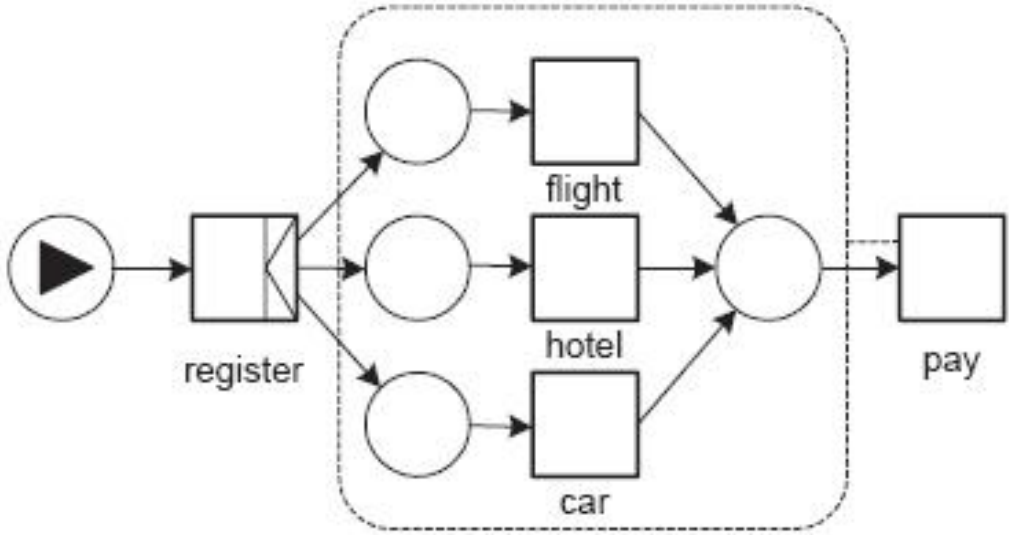


Figura 3.7: implementazione del pattern Discriminator (pattern 9)

Questi esempi dimostrano che YAWL riesce a gestire molte situazioni critiche che altri workflow non riuscirebbero a gestire. Dei 20 pattern discussi precedentemente YAWL ne supporta 19. L'unico non supportato è la terminazione implicita (Pattern 11), ma questa, più che una carenza, è stata una scelta che forza il progettista a pensare alle proprietà di terminazione del workflow. Infatti sarebbe molto semplice estendere YAWL con questo pattern (basterebbe infatti connettere semplicemente tutte le condizioni di output con un OR-join che abbia una condizione di output nuova e unica). La terminazione implicita nasconde inoltre errori di progettazione perché non permette di identificare alcune situazioni di stallo.

In questa ultima parte del paragrafo tratteremo il ruolo del tempo e degli eventi. Ovviamente un linguaggio di workflow deve essere in grado di rappresentare la ricezione di eventi e i costrutti basati sul tempo, come i time-out. Non c'è bisogno di aggiungere elementi specifici per rappresentare questi costrutti, ma semplicemente useremo *task* che possano gestire queste cose.

Ad esempio possiamo creare vari tipi di *task* temporizzati, ovvero *task* che non fanno altro che aspettare time-out, oppure *task* per eventi. Useremo la lettera 'T' per identificare i primi e la 'E' per i secondi. Negli esempi che seguono il *task* di pagamento (**pay**) è un *task* per eventi, ovvero un *task* che attende un particolare evento (in questo caso un messaggio di pagamento), mentre quello temporizzato è un time-out, ovvero un *task* in attesa che un determinato periodo di tempo sia completo. Ovviamente il time-out può essere creato in modo che aspetti una determinata quantità di tempo (ad esempio un'ora) oppure uno specifico momento (le ore 16.30 di oggi).

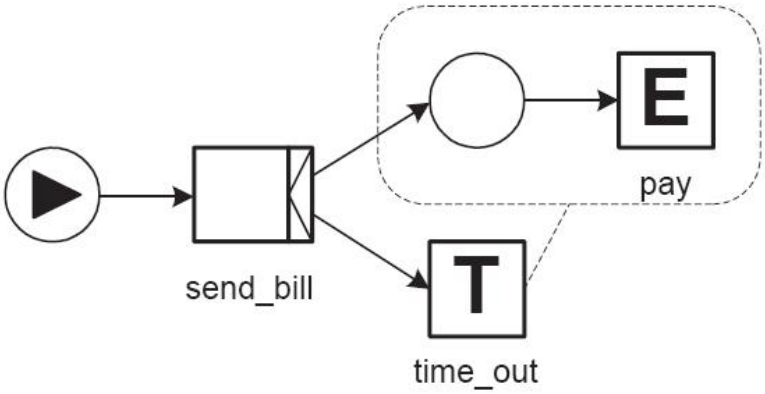


Figura 3.8: esempio di cancellazione di un task, anche durante l'esecuzione

La figura mostra il primo esempio di utilizzo di tale strutture. In questo caso quando il time-out termina l'attesa cancella il processo di pagamento. Da notare che, anche se il pagamento è stato processato, il ramo in alto viene cancellato. Da notare anche che la condizione tra **send_bill** e **pay** è mostrata esplicitamente: in questo modo è possibile indicare chiaramente che, allo scadere del time-out, anche i pagamenti in attesa vengono cancellati..

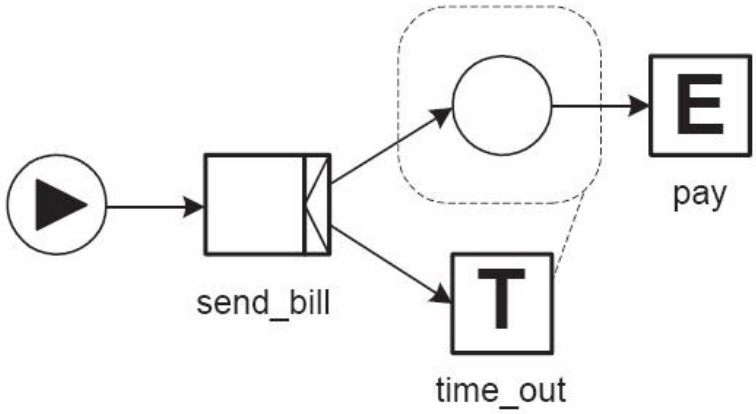


Figura 3.9: esempio di cancellazione di un task in attesa di esecuzione

Questo esempio mostra la situazione in cui il time-out cancella il processo di pagamento solo quando non è ancora iniziato. Nei due esempi precedenti non è necessario cancellare il processo di time-out dopo il pagamento perché se il time-out scade dopo il pagamento non ci sarà nulla da cancellare. Ma se il time-out non cancella solo token ma esegue un lavoro reale o manda in esecuzione altri *task* che possono eseguire una ulteriore gestione della cancellazione, allora il *task* **pay** deve cancellare tutta la parte collegata al time-out, come mostrato di seguito.

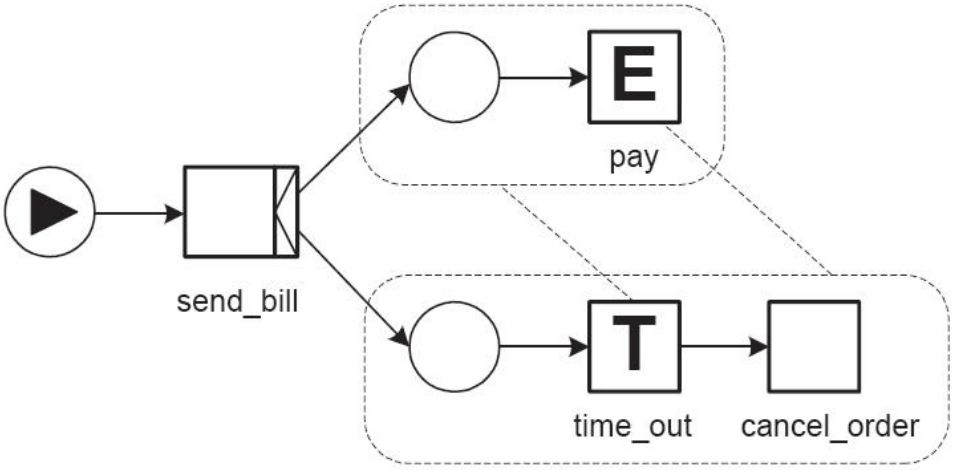


Figura 3.10: esempio di cancellazione reciproca tra due task

Il *task* **cancel_order** è eseguito dopo che si è verificato un time-out e serve per

compensare l'effetto di non aver ricevuto il pagamento entro il tempo specificato. Quando il pagamento viene effettuato per tempo cancella l'intero ramo sottostante. Da notare che il time-out ha una condizione di input esplicita a indicare che se, per una qualche ragione, il *task* non è stato fatto partire quando il pagamento è completo, il processo di cancellazione viene ugualmente disattivato.

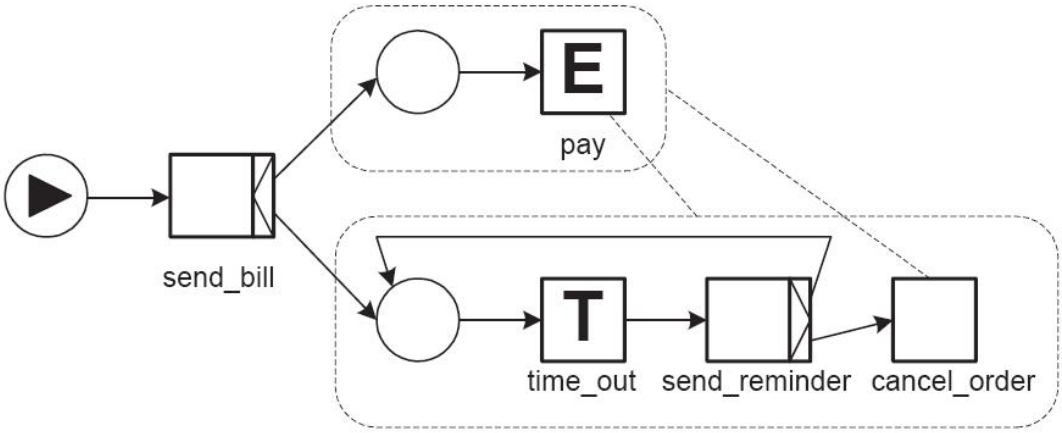


Figura 3.11: esempio di implmentazione di un reminder

L'ultimo esempio è una estensione del precedente in cui però si è aggiunta la possibilità di mandare delle notifiche prima di cancellare il pagamento. Lo XOR-split permette di eseguire l'invio di queste notifiche più di una volta prima di cancellare l'ordine. In questo modo si può modellare un comportamento tipo quello descritto dal seguente testo: "*se l'acquirente non paga entro una settimana, viene mandata una notifica di mancato pagamento. Questa cosa viene ripetuta ogni settimana per quattro volte o finché l'acquirente non paga. Alla quarta notifica, se l'acquirente non ha pagato, l'ordine viene cancellato*".

3.6 Il linguaggio XML per un workflow YAWL

Sono stati sviluppati due software che permettono la creazione e l'esecuzione di workflow YAWL: l'editor (il cui funzionamento è descritto in [5]), che permette di

disegnare il workflow e impostarne le caratteristiche (dati, condizioni e tutto ciò che fa parte del linguaggio YAWL precedentemente descritto), e l'engine, che permette di simulare l'esecuzione del workflow generato dall'editor. Per questa tesi sono stati usati l'editor versione *1.4* e l'engine versione *5.01*.

Siccome il progetto svolto per la tesi riguarda il linguaggio usato per scambiare le descrizioni dei workflow tra il programma editor di YAWL e il suo motore di esecuzione, in questo paragrafo analizzeremo gli elementi che fanno parte proprio di questo linguaggio, che è basato su XML.

Intanto un documento XML per la specifica di un workflow YAWL è formato da una serie di elementi. Per dare una spiegazione più dettagliata prenderemo in esame il documento XML che rappresenta lo scenario proposto in [5]. Lo scenario riguarda le possibilità di cariera che ha uno studente all'uscita dalle scuole superiori: può entrare nell'università e completare i suoi studi oppure continuare con studi privati che eventualmente lo porteranno ad un lavoro.

Qui di seguito riporto il workflow finale (per i dettagli si rimanda a [5]):

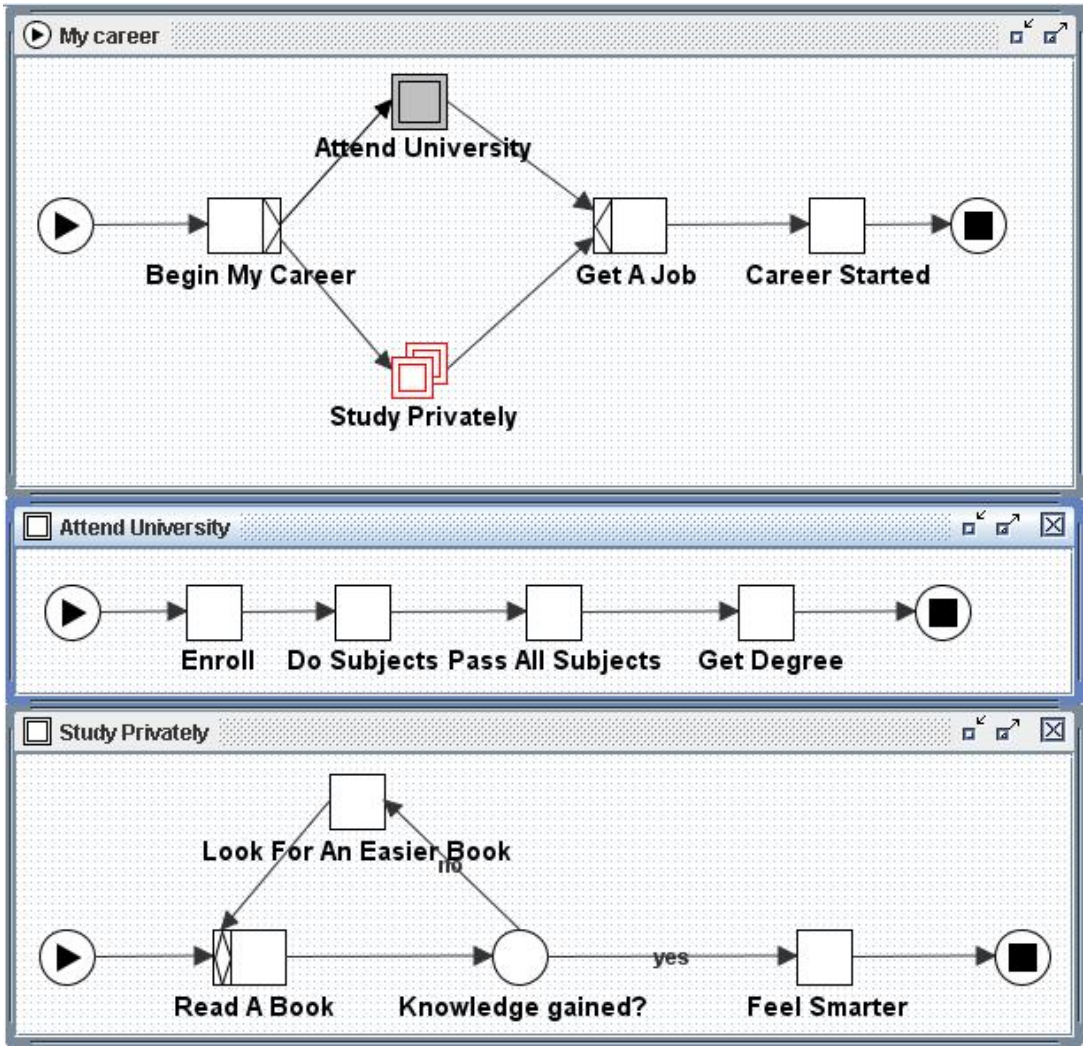


Figura 3.12: il workflow dell'esempio preso in esame

La figura *3.12* mostra il workflow: all'inizio della carriera si può scegliere se frequentare l'università (nel qual caso non ci sarà tempo per intraprendere studi privati, quindi il *task* composto con istanze multiple **Study Privately** farà parte del set di cancellazione di **Attend University**) o intraprendere studi privati. In entrambi i casi, al termine di queste attività si otterrà un lavoro e inizierà la propria carriera.

Le due sottoreti **Attend University** e **Study Privately** mostrano rispettivamente come si svolge la frequentazione dell'università e degli studi privati, e costituiscono le decomposizioni degli omonimi *task* compositi della rete radice.

Inoltre **Study Privately** è un task che permette istanze multiple. Nonostante non

sia visibile dalla figura, il numero di istanze da eseguire va da 5 a 100, la soglia di completamento è 50 e le istanze sono create staticamente (seguendo la notazione degli esempi precedenti si sarebbe dovuto porre sopra al task la scritta '**[5, 100, 50, static]**').

Il file di specifica per l'engine di YAWL inizia con l'elemento che determina quale è il tipo di file (ovvero XML), la sua versione e il tipo di codifica, così come vengono definiti tutti i file XML.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Dopo si definisce l'elemento radice di tutto il documento YAWL. Gli attributi di questo elemento determinano i *namespace* utilizzati e la versione del documento stesso (che è relativa alla versione dell'editor utilizzata).

```
<specificationSet xmlns="http://www.citi.qut.edu.au/yawl"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   version="Beta 7.1"
   xsi:schemaLocation="http://www.citi.qut.edu.au/yawl
                           d:/yawl/schema/YAWL_SchemaBeta7.1.xsd">
```

Il figlio principale dell'elemento radice dichiara la specifica del workflow. Tale elemento, **`specification`**, ha come unico attributo la locazione fisica del file di descrizione YAWL.

```
<specification uri="unnamed.ywl">
```

Gli elementi figli di **`specification`** sono di 3 tipi:

- **`metaData`** che mantiene informazioni relative all'autore del workflow, alla versione di questo e altri dati utili. Può esserci un solo elemento **`metaData`** in una **`specification`**.

- **schema** che mantiene la definizione di tipi di variabile complessi, descritti tramite XMLSchema e basati sui tipi di variabile base di YAWL (stringhe, interi, booleani…)
- **decomposition** che descrive i dati di una (sotto) rete o di un template per i *task*.

Il primo elemento, **metaData**, generato per il workflow visto è così composto:

```
<metaData>
  <title>My career</title>
  <creator>Andrea Lorenzani</creator>
  <description>Descrizione del workflow</description>
  <validFrom>2006-03-07</validFrom>
  <validUntil>2006-08-07</validUntil>
  <version>0.1</version>
</metaData>
```

Il significato dei vari elementi è evidente: **title** specifica un titolo per il workflow, **creator** indica chi è l'autore del workflow, **description** contiene una breve descrizione del workflow, **validFrom** e **validUntil** mantengono le date di inizio e fine validità del workflow, **version** permette di indicare quale è la versione di questo workflow.

Da notare che questi dati sono stati introdotti dalla versione 1.4 dell'editor YAWL.

Esaminiamo ora l'elemento **schema** per il workflow visto:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="Geek">
   <sequence>
    <element name="Name" type="string" />
    <element name="Salary" type="double" />
   </sequence>
  </complexType>
</schema>
```

Nel nostro esempio veniva infatti definito un tipo di variabile composta chiamata '**Geek**' che era una sequenza di due elementi: una stringa di nome **Name** e un double di nome **Salary**. Nel seguito vedremo che verrà usata una variabile di questo tipo.

Da notare che l'elemento **schema** definisce il *namespace* per l'XMLSchema, visto che le definizioni useranno proprio questo tipo di linguaggio. In più anche l'elemento **schema** è unico.

Per quanto riguarda invece l'elemento **decomposition**, questo viene utilizzato per specificare o una rete (o una sottorete) oppure dati relativi alla decomposizione di uno o più *task*. Un attributo, **xsi:type** (notare l'appartenenza al *namespace* dello schema di YAWL), permette di capire immediatamente a quale dei due casi ci si sta riferendo; se il suo valore è **NetFactsType** l'elemento conterrà la specifica di una (sotto) rete, se invece è **WebServiceGatewayFactsType** sarà la decomposizione di uno o più *task*. Nel caso di una rete l'attributo **isRootNet="true"** (opzionale) indica se tale rete è la radice del workflow o no. Chiaramente una sola rete può essere radice di un workflow.

Un elemento può essere padre di altri elementi, in base al significato che ha la decomposizione stessa. Gli elementi che ne possono far parte sono:

- **inputParam**, **outputParam** e **localVariable** che definiscono rispettivamente una variabile di input, una di output o una locale per un *task* o per una (sotto) rete; una variabile può apparire contemporaneamente solo come input e come output;
- **processControlElements** che descrive il workflow di una (sotto) rete in termini di *task* e flussi di controllo;
- **yawlService** mantiene i dati per l'invocazione, tramite YAWL, di un *web service*.

Ovviamente tutti questi elementi sono opzionali e spesso nel nostro workflow ci saranno delle decomposizioni vuote che servivano soltanto a dare un nome al *task* che fa parte del workflow. Continuiamo ora con l'esempio preso in esame per esaminare la sintassi di questi elementi:

```
 <decomposition id="My_career" isRootNet="true"
xsi:type="NetFactsType">
```

```
<localVariable>
      <name>testGeek</name>
      <type>Geek</type>
      <namespace>http://www.w3.org/2001/XMLSchema</namespace>
      <initialValue>
            <Name>Lindsay</Name> <Salary>6000.00</Salary>
      </initialValue>
</localVariable>
```

Come possiamo vedere dalla prima riga riportata sopra del documento, stiamo considerando una rete (**xsi:type="NetFactsType"**), e per la precisione quella a radice dell'albero del workflow (**isRootNet="true"**). Per questa rete è stata definita una variabile locale tramite l'elemento **localVariable** che è composto da quattro elementi figli: **name** indica il nome della variabile, ovvero il suo identificativo (unico), **type** indica il tipo di variabile (in questo caso è di tipo **Geek**, ovvero quel tipo composito che abbiamo definito noi e la cui descrizione è nell'elemento **schema**), **namespace** che indica il *namespace* di riferimento della variabile e **initialValue** che permette di definirne un valore iniziale.

```
<processControlElements>
  <inputCondition id="InputCondition_1">
         <flowsInto>
                <nextElementRef id="Begin_My_Career_5" />
         </flowsInto>
    </inputCondition>
  <task id="Begin_My_Career_5">
     <flowsInto>
                <nextElementRef id="Attend_University_6" />
                <predicate ordering="0">true()</predicate>
         </flowsInto>
     <flowsInto>
                <nextElementRef id="Study_Privately_7" />
                <isDefaultFlow />
         </flowsInto>
         <join code="xor" />
```

```
            <split code="xor" />
            <decomposesTo id="Begin_My_Career" />
      </task>
      <task id="Attend_University_6">
         <flowsInto>
                     <nextElementRef id="Get_A_Job_4" />
            </flowsInto>
            <join code="xor" />
            <split code="and" />
            <removesTokens id="Study_Privately_7" />
         <removesTokensFromFlow>
                     <flowSource id="Begin_My_Career_5" />
                     <flowDestination id="Study_Privately_7" />
            </removesTokensFromFlow>
            <decomposesTo id="Attend_University" />
      </task>
      <task id="Study_Privately_7">
         <flowsInto>
                     <nextElementRef id="Get_A_Job_4" />
            </flowsInto>
            <join code="xor" />
            <split code="and" />
         <startingMappings>
               <mapping>
                              <expression query="/My_career/geek" />
                              <mapsTo>geek</mapsTo>
                     </mapping>
            </startingMappings>
         <completedMappings>
               <mapping>
                              <expression
query="/Study_Privately/geek/*" />
                               <mapsTo>testGeek</mapsTo>
                     </mapping>
            </completedMappings>
            <decomposesTo id="Study_Privately" />
      </task>
      <task id="Get_A_Job_4">
```

```
      <flowsInto>
                  <nextElementRef id="Career_Started_3" />
            </flowsInto>
            <join code="xor" />
            <split code="and" />
            <decomposesTo id="Get_A_Job" />
   </task>
   <task id="Career_Started_3">
      <flowsInto>
                  <nextElementRef id="OutputCondition_2" />
            </flowsInto>
            <join code="xor" />
            <split code="and" />
            <decomposesTo id="Career_Started" />
   </task>
   <outputCondition id="OutputCondition_2" />
</processControlElements>
</decomposition>
```

Questa parte del documento descrive il workflow vero e proprio; **processControlElements** è composto da due tipi di elemento: **condition** (tra cui una **inputCondition** e una **outputCondition**) e **task**, che ovviamente descrivono condizioni e *task*. Entrambi questi elementi hanno un attributo **id** che li distingue univocamente e contengono (ad eccezione di **outputCondition**) uno o più elementi **flowsInto** che ne descrivono i collegamenti per il flusso di controllo tramite il loro figlio **nextElementRef**, che ha come attributo l'**id** del *task* o della condizione a cui è collegato.

Se le condizioni specificano solo questi elementi generici, nei *task* si possono trovare altre informazioni: **join** e **split** indicano il tipo di join e di split, il cui significato è spiegato nei paragrafi precedenti, attraverso l'attributo **code**.

In caso di un *task* con uno split di tipo OR o XOR, per determinare quale dei flussi riceverà un token si usa un predicato (elemento **predicate** figlio dell'elemento **flowsInto**) che contiene una espressione XPath da verificare per seguire questo determinato flusso. Il campo **ordering**, che fa parte dell'elemento **predicate**, dà un

ordinamento, e quindi una priorità, a questi flussi, utile in caso si utilizzi uno XOR-split.

Il funzionamento di OR e XOR split è quindi il seguente: nel caso di XOR-split verrà mandato un token all'elemento con **`ordering`** più basso tra quelli il cui campo **`predicate`** risulta essere vero, e se nessun **`predicate`** risulta verificato, il token verrà mandato comunque nel flusso che ha l'elemento che lo contraddistingue come flusso di default, ovvero l'elemento **`isDefaultFlow`** (figlio anche questo dell'elemento **`flowsInto`**). L'OR-split invece manderà un token a tutti i flussi il cui campo **`predicate`** risulta essere verificato, e nel caso nessuno lo fosse verrà mandato al flusso di default, ovvero quello con l'elemento **`isDefaultFlow`**.

Come spiegato nel *paragrafo 3.4* i dati contenuti in un workflow YAWL possono essere scambiati solo tra i *task* e la rete di cui fanno parte, ma non direttamente tra i *task* stessi. Gli elementi **`startingMappings`** e **`completedMappings`** hanno lo scopo di definire quali sono questi scambi. Il primo di questi due elementi indica quali sono i valori assegnati quando il *task* sta per essere eseguito (quindi assegna le variabili di input), il secondo quali valori sono assegnati al suo completamento (quindi assegna quelle di output). Ognuno di questi elementi è composto da uno o più **`mapping`** che, a sua volta, è formato da una **`expression`** che, con una espressione XPath, indica dove verrà preso il valore da mettere nella variabile segnalata dall'elemento **`mapsTo`**.

Un altro elemento di un *task* è **`removesTokens`** che indica quale è il *cancellation set* (l'insieme di *task* e condizioni a cui verrà applicato il pattern di cancellazione numero 19). Il suo attributo **`id`** indica a quale *task* debbano essere cancellati i token. Per rimuovere un token da un flusso, invece che da un *task*, si usa l'elemento **`removesTokensFromFlow`**: siccome i flussi non sono identificati univocamente, si usano due sottoelementi figli per identificare la sorgente (**`flowSource`**) e la destinazione (**`flowDestination`**) del flusso da cui si vogliono eliminare eventuali token.

Per finire i *task* possono avere associata una **`decomposesTo`** che indica a quale **`decomposition`** sono associati. Questa, come già visto, può specificare una sottorete (nel qual caso il *task* deve essere stato composito), le variabili associate al *task* e

determina il nome del *task* stesso.

```
<decomposition id="Get_A_Job" xsi:type="WebServiceGatewayFactsType"
/>
<decomposition id="Do_Subjects" xsi:type="WebServiceGatewayFactsType"
/>
<decomposition id="Get_Degree" xsi:type="WebServiceGatewayFactsType"
/>
<decomposition id="Look_For_An_Easier_Book"
                xsi:type="WebServiceGatewayFactsType" />
<decomposition id="Pass_All_Subjects"
xsi:type="WebServiceGatewayFactsType" />
```

Come possiamo notare tutte queste **decomposition** definiscono solo il nome dei vari *task*. Né variabili né tantomeno (sotto) reti sono state definite tramite questi elementi.

```
<decomposition id="Study_Privately" xsi:type="NetFactsType">
  <inputParam>
          <name>geek</name>
          <type>Geek</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </inputParam>
  <outputParam>
          <name>geek</name>
          <type>Geek</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
  </outputParam>
  <processControlElements>
          <inputCondition id="InputCondition_22">
                  <flowsInto>
                          <nextElementRef id="Read_A_Book_26" />
                  </flowsInto>
          </inputCondition>
          <task id="Read_A_Book_26">
                  <flowsInto>
```

```
                        <nextElementRef id="Knowledge_gained_24"
/>
                </flowsInto>
                <join code="or" />
                <split code="and" />
                <decomposesTo id="Read_A_Book" />
        </task>
        <condition id="Knowledge_gained_24">
                <flowsInto>
                        <nextElementRef id="Feel_Smarter_27" />
                </flowsInto>
                <flowsInto>
                        <nextElementRef
id="Look_For_An_Easier_Book_25" />
                </flowsInto>
        </condition>
        <task id="Look_For_An_Easier_Book_25">
                <flowsInto>
                        <nextElementRef id="Read_A_Book_26" />
                </flowsInto>
                <join code="xor" />
                <split code="and" />
                <decomposesTo id="Look_For_An_Easier_Book" />
        </task>
        <task id="Feel_Smarter_27">
                <flowsInto>
                        <nextElementRef id="OutputCondition_23"
/>
                </flowsInto>
                <join code="xor" />
                <split code="and" />
                <decomposesTo id="Feel_Smarter" />
        </task>
        <outputCondition id="OutputCondition_23" />
  </processControlElements>
  </decomposition>
```

Questa **decomposition** descrive la sottorete **Study_Privately**. La sintassi è già stata analizzata. Qui è possibile notare un esempio di variabile sia di input che di output (**geek**, tra l'altro di tipo definito nello **schema**) e come l'elemento **flowsInto** venga replicato in un *task* o, come in questo caso, in una condition per ogni flusso uscente. Per il resto, se esaminate il workflow nella figura, sarà facile rendersi conto che non ci sono elementi particolari.

Il resto del documento, riportato di seguito, conclude la definizione del workflow mostrato nella figura *3.12*.

```
<decomposition id="Begin_My_Career"
xsi:type="WebServiceGatewayFactsType" />
<decomposition id="Feel_Smarter"
xsi:type="WebServiceGatewayFactsType" />
<decomposition id="Attend_University" xsi:type="NetFactsType">
   <localVariable>
                  <name>StudentNumber</name>
                  <type>string</type>

         <namespace>http://www.w3.org/2001/XMLSchema</namespace>
                  <initialValue />
   </localVariable>
   <localVariable>
         <name>SubjectCode</name>
         <type>string</type>
         <namespace>http://www.w3.org/2001/XMLSchema</namespace>
         <initialValue />
   </localVariable>
   <processControlElements>
      <inputCondition id="InputCondition_13">
            <flowsInto>
                        <nextElementRef id="Enroll_15" />
                  </flowsInto>
         </inputCondition>
      <task id="Enroll_15">
            <flowsInto>
                        <nextElementRef id="Do_Subjects_16" />
```

```
            </flowsInto>
            <join code="xor" />
            <split code="and" />
        <startingMappings>
            <mapping>
                <expression
                    query="<StudentNumber>{/Attend_Universit
                    y/StudentNumber/text()}</StudentNumber>"
                    />
                        <mapsTo>StudentNumber</mapsTo>
                    </mapping>
            <mapping>
                <expression
                    query="<SubjectCode>{/Attend_University/
                    SubjectCode/text()}</SubjectCode>" />
                        <mapsTo>SubjectCode</mapsTo>
                    </mapping>
            </startingMappings>
        <completedMappings>
            <mapping>
                        <expression
                    query="<StudentNumber>{/Enroll/SubjectCo
                    de/text()}</StudentNumber>" />
                        <mapsTo>StudentNumber</mapsTo>
                    </mapping>
            <mapping>
                        <expression

    query="<SubjectCode>{/Enroll/SubjectCode/text()}
                    </SubjectCode>" />
                        <mapsTo>SubjectCode</mapsTo>
                    </mapping>
            </completedMappings>
            <decomposesTo id="Enroll" />
    </task>
<task id="Do_Subjects_16">
    <flowsInto>
                    <nextElementRef id="Pass_All_Subjects_18"
```

```
/>
                    </flowsInto>
                    <join code="xor" />
                    <split code="and" />
                    <decomposesTo id="Do_Subjects" />
            </task>
        <task id="Pass_All_Subjects_18">
              <flowsInto>
                              <nextElementRef id="Get_Degree_17" />
                    </flowsInto>
                    <join code="xor" />
                    <split code="and" />
                    <decomposesTo id="Pass_All_Subjects" />
            </task>
        <task id="Get_Degree_17">
              <flowsInto>
                              <nextElementRef id="OutputCondition_14"
/>
                    </flowsInto>
                    <join code="xor" />
                    <split code="and" />
                    <decomposesTo id="Get_Degree" />
            </task>
            <outputCondition id="OutputCondition_14" />
      </processControlElements>
</decomposition>
<decomposition id="Career_Started"
xsi:type="WebServiceGatewayFactsType" />
<decomposition id="Enroll" xsi:type="WebServiceGatewayFactsType">
   <inputParam>
          <name>StudentNumber</name>
          <type>string</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
   </inputParam>
   <inputParam>
          <name>SubjectCode</name>
          <type>string</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
```

```
    </inputParam>
    <outputParam>
          <name>StudentNumber</name>
          <type>string</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
    <outputParam>
          <name>SubjectCode</name>
          <type>string</type>
          <namespace>http://www.w3.org/2001/XMLSchema</namespace>
    </outputParam>
  </decomposition>
  <decomposition id="Read_A_Book" xsi:type="WebServiceGatewayFactsType"
  />
  </specification>
  </specificationSet>
```

Con questo si conclude il documento per l'engine YAWL creato dall'editor nel caso del workflow visto nella figura. In questo esempio, tratto da [5] come già detto, non sono state definite interazioni con *web service* esterni, cosa che è possibile fare con l'engine di YAWL.

Per esaminare brevemente la sintassi di tale funzione riporto una parte di un documento XML di specifica per un workflow YAWL che fa una chiamata ad un *web service* esterno.

```
<yawlService id="http://localhost:8080/yawlWSInvoker/">
  <wsdlLocation>http://www.abundanttech.com/webservices/bnprice/bnprice
  .wsdl
  </wsdlLocation>
  <operationName>GetBNQuote</operationName>
</yawlService>
```

Infatti l'editor permette di specificare la locazione di una istanza dell'engine in esecuzione in modo da essere avvantaggiata dalle impostazioni di quella specifica

istanza. In pratica se l'istanza di un engine può connettersi con determinati *web service* tramite l'editor possiamo far sì che all'esecuzione di un *task* corrisponda una chiamata a uno dei *web service* connessi con l'engine.

Il campo **`id`** indica quindi quale sarà il *web service* invocato, **`wsdlLocation`** indica quale è la locazione del documento WSDL per tale *web service* e **`operationName`** quale sarà l'operazione invocata.

Capitolo 4 - BPEL2YAWL

In questo capitolo esamineremo a fondo come è stata sviluppata la tesi, il cui obiettivo è lo sviluppo di una metodologia di traduzione che permetta di ottenere da un processo BPEL4WS una sua rappresentazione mediante YAWL. La rappresentazione YAWL deve essere tale che, eseguita nelle stesse condizioni del processo originale (stessi valori delle variabili, stessi messaggi ricevuti, stessi errori generati…), simuli esattamente lo stesso comportamento di questo mandando in esecuzione i task che corrispondono alle attività BPEL4WS che sarebbero state mandate in esecuzione dal processo. Vedremo che ad una singola attività corrisponderanno più task di un workflow YAWL. L'analisi della progettazione di questo traduttore sarà il tema del paragrafo *4.1*.

Una volta che è stato creato uno 'schema di traduzione' è stato possibile sviluppare un programma, scritto in JAVA, che eseguisse in maniera automatica questa traduzione. Nel paragrafo *4.2* viene presentata ad alto livello l'implementazione di questo traduttore.

Infine nell'ultimo paragrafo di questo capitolo verrà presentato un esempio del funzionamento di tale traduttore.

4.1 Progettazione

Per quanto riguarda la progettazione del traduttore è importante sottolineare che lo scopo principale era avere una rappresentazione fedele del processo. Non essendo questa una cosa banale, per catturare tutte le particolarità del linguaggio BPEL4WS abbiamo sviluppato un tipo di traduzione che fosse formata da 'blocchi' o pattern specifici per ogni parte del processo e tali che, per ottenere la traduzione, potessero essere istanziati e collegati tra loro secondo la logica di come comparivano nel processo BPEL4WS.

Quindi, come sarà ampiamente discusso in seguito, ogni attività tradotta comporta l'introduzione nel workflow di almeno un pattern (discuteremo nel paragrafo *4.1.1* dei pattern di traduzione) se questa è una attività semplice da rappresentare, o più di uno per le attività più complesse, e questo pattern verrà collegato nel workflow nell'esatto punto in cui ci si dovrebbe aspettare che venga eseguito.

In questo modo, partendo dalla descrizione del processo del documento BPEL4WS, e usando questi 'blocchi' che rappresentano attività, gestori e più in generale tutti gli elementi basilari di BPEL, è possibile ricomporre un processo in YAWL. In più i pattern gestiscono tutte le possibilità di esecuzione di tali elementi minimali, e questo permette di avere sempre delle rappresentazioni di processi che simulino tutti i possibili comportamenti di BPEL4WS; tale era infatti lo scopo principale di questa traduzione.

4.1.1 Gli schemi di traduzione

Per catturare ogni dettagli della specifica BPEL4WS è stato necessario analizzare bene il ruolo che può avere ogni singola *attività* all'interno di un processo BPEL4WS. Infatti, sempre in relazione a quanto detto nel capitolo *2*, una attività ha un significato dal punto di vista della sua esecuzione, nel senso che ogni tipo di *attività* svolge un compito diverso, ma tutte queste possono avere anche degli altri comportamenti:

- possono dover essere saltate perché contenute all'interno di una *attività* composita che deve essere saltata o a causa dei *link* in ingresso o per la *dead-path elimination*;
- possono dover essere saltate a causa dei propri *link* in ingresso;
- possono generare un errore in seguito alla loro esecuzione;
- possono generare un errore di join (attributo `suppressJoinFailure` settato a `no` e valore **false** della condizione `joinCondition`, come spiegato nel paragrafo *2.6.1*).

In più è stato possibile sviluppare un metodo di traduzione di questo tipo perché BPEL4WS usa *attività* strutturate per specificare in quale ordine le *attività* debbano

essere eseguite. Per fare un esempio, la seconda *attività* di una **sequence** può essere eseguita solo quando la prima ha finito la sua esecuzione. In più il costrutto **flow**, come abbiamo detto nel capitolo *2*, permette di definire dei collegamenti di sincronizzazione (i *link*) tra le *attività*.

Queste considerazioni costituiscono il punto di partenza per la creazione di un pattern, ovvero una struttura standard composta di task, che rappresenti un singolo elemento della traduzione generale. Tutti i pattern di traduzione saranno varianti del pattern mostrato nella figura seguente:

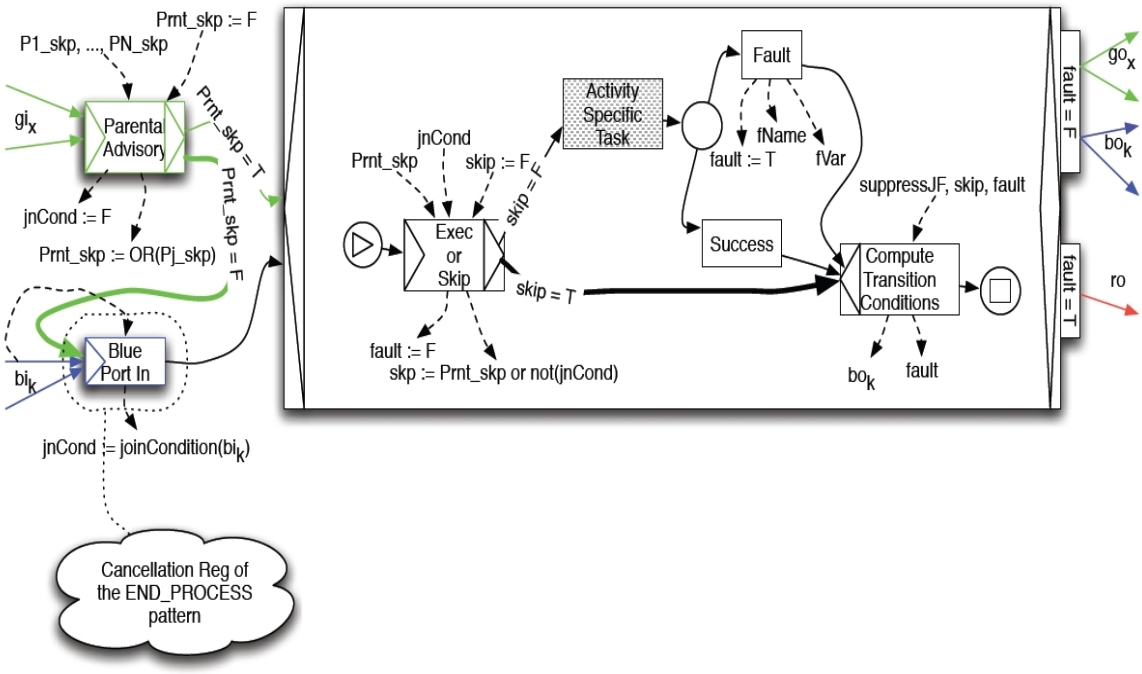


Figura 4.1: il pattern generico di traduzione

Per capire meglio il significato di questo elemento basilare della nostra traduzione, ne verrà spiegato il significato parte per parte. Intanto la prima cosa fondamentale di questo pattern è la colorazione dei flussi. Si usano tre colori:

- i flussi verdi rappresentano le dipendenze strutturali, che indicano l'ordine in cui si

svolgerà l'esecuzione dell'*attività* in base alla struttura delle altre *attività* del processo. Per esempio, le *attività* interne ad una **`sequence`** devono essere eseguite in sequenza quindi la prima manderà il suo flusso verde alla seconda, che lo manderà alla terza e così via. Quando nei pattern è presente il termine "**gi**" si riferisce ad un **green input**, ovvero ad un flusso strutturale in ingresso, con "**go**" invece si indica un **green output**, un flusso strutturale in uscita;

- i flussi blu indicano i collegamenti per quanto riguarda i *link* tra le *attività*. Il *link* trasmette inoltre il valore del proprio status, che verrà utilizzato per il calcolo dell'espressione **`joinCondition`** nell'*attività* destinazione. Anche in questo caso "**bi**" indica un flusso di *link* in ingresso e "**bo**" uno in uscita;

- i flussi rossi, per finire, indicano i collegamenti per i fallimenti. Le attività hanno solo i **red output** (se possono generare un errore) mentre i **`faultHandler`** ricevono **red input** e possono, nelle loro *attività* interne, generare **red output**. Inoltre quando viene seguito un flusso rosso vengono prima determinati i valori delle variabili necessarie a gestirlo (ovvero le variabili **fName** e **fVar**).

Generalmente (fa eccezione solo la **`terminate`**) i token che vengono propagati da un pattern sono quelli verdi e blu se il pattern è stato eseguito senza sollevare eccezioni, altrimenti vengono mandati solo quelli rossi.

Una volta che saranno stabilite tutte le varianti di questo pattern generico, che corrispondono, come vedremo, a parti di *attività* e a costrutti di BPEL4WS, per tradurre un documento basterà istanziare tutti i pattern relativi ai suoi elementi e collegarli tra loro in base ai colori dei loro flussi (quindi il **green output** di un pattern con il **green input** del pattern successivo e così via).

Per quanto riguarda i task presenti nel pattern ce ne sono tre principali: il **Parental Advisory**, il **Blue Port In** e il **Main Task**, il quale è un task composito che ha al suo interno la sottorete mostrata in figura. Il significato di tutti i task della figura è il seguente:

- il **Parental Advisory** ha il compito di stabilire se l'*attività* che si sta rappresentando deve essere saltata per via del fatto che il suo "genitore" (ovvero l'*attività* che la

contiene) è stato saltato a sua volta. Questo come dicevamo prima può essere causato solo per via dei *link* dell'*attività* che la contiene. Le variabili in ingresso **P1_skp, … , PN_skp** stanno a significare che ovviamente tutti i livelli di annidamento superiori all'*attività* in esame possono causare il salto dell'*attività*, ma dal punto di vista pratico ovviamente basta sapere solo se il livello di annidamento direttamente superiore all'*attività* (cioè l'*attività* padre) deve essere saltata per causare il salto dell'*attività* figlia, indipendentemente dal fatto se il padre è dovuto essere saltato perché contenuto in una *attività* da saltare o perché ha provocato da solo il salto a causa della propria `joinCondition`. Il **Parental Advisory** segnala alla fine dell'esecuzione quale deve essere il comportamento da tenere a riguardo del salto dell'*attività*, e in base a questo risultato sceglie anche quale dei due task ad esso collegato mandare in esecuzione: se l'*attività* non deve essere saltata a priori si provvederà a valutare lo stato di eventuali *link* attraverso il **Blue Port In**, altrimenti si passerà subito il flusso al **Main Task**;

- il **Blue Port In** rimane in attesa, a causa dell'AND-join, di tutti i *link* in ingresso, proprio come ci si aspetta da una *attività* destinazione di *link*. Ovviamente se non ci sono *link* in ingresso l'attività viene subito eseguita. Il suo unico compito è quello di valutare l'espressione `joinCondition` dell'*attività* che rappresenta;

- il task **Exec or Skip** decide se l'*attività* specifica di questa *attività* BPEL4WS debba essere eseguita o saltata. Infatti, attraverso il **Main Task** che contiene questa rete, l'**Exec or Skip** ottiene come input il valore calcolato dal **Parental Advisory** e quello del **Blue Port In**: se l'*attività* padre è stata saltata oppure non è stata verificata la `joinCondition` allora il task che rappresenta questa specifica *attività* non dovrà essere eseguito, e si passerà comunque al task **Compute Transition Condition**. Altrimenti il flusso passerà alla **Activity Specific Task**. Comunque nella variabile **skip** verrà tenuto il valore del salto, che verrà riutilizzato nella **Compute Transition Condition**;

- il task **Activity Specific Task** di volta in volta assumerà un significato diverso in base al tipo di *attività* BPEL4WS che questo pattern rappresenta. Eseguire questo task nella simulazione di YAWL equivale ad eseguire una parte (generalmente una *attività*, ma non sempre, come per esempio nel caso della **`assign`** o delle *attività* composite) del processo BPEL4WS. Se quella parte di processo può generare un fallimento, questo task invierà il flusso a una **`condition`** come mostrato in figura. In questo caso sarà possibile, durante la simulazione, scegliere se l'attività ha dato esito positivo (mandando il flusso al task **Success**) o negativo (mandandolo al task **Fault** che provvederà a salvare i dati importanti quali **`faultName`** e **`faultVariable`** come descritto nel paragrafo *2.5.5* e ad indicare tramite la variabile **fault** che si è verificato un errore). Questo "deferred choice" (ovvero la **`condition`** collegata ai due task **Success** e **Fault**, vedi il pattern 16 al paragrafo *3.2*) si usa per le *attività* che possono generare errori, altrimenti l'**Activity Specific Task** sarà collegato direttamente alla **Compute Transition Condition**;

- il task **Compute Transition Condition** calcola il valore di **fault** (ci sarà un fallimento se si è passati dal task **Fault** descritto prima oppure se non è stata verificata la **`joinCondition`** nel **Blue Port In** e la **`suppressJoinFailure`** ha come valore **`no`**) che serve per decidere quale direzione prenderà il flusso (**fault='false'** vuol dire assenza di errori, quindi verranno mandati i token verdi e blu, **fault='true'** invece indica la presenza di errore e verrà mandato il token rosso) e la sua funzione dovrebbe essere quella di calcolare i valori dei **`LinkStatus`** da utilizzare per le *attività* che sono destinazione di *link* che hanno come sorgente questo pattern. In realtà poi, per il progetto, questi valori saranno valutati tramite i *mapping* (di cui abbiamo parlato nel paragrafo *3.4* e nel paragrafo *3.6*) di completamento del **Main Task**.

Questo è sostanzialmente il funzionamento di una singola parte della traduzione e

costituisce il fondamento per realizzare tutta la traduzione. Vedremo che in linea generale lo schema riportato sopra non subirà quasi mai grosse variazioni, e che ci sono dei pattern derivati da quello generico che si ripropongono negli schemi di traduzione di molte *attività*.

Come detto nella parte introduttiva di questo capitolo, una volta definiti tutti i pattern per tutte le *attività* potremo, per tradurre un processo BPEL, creare per ogni *attività* una istanza del suo pattern di traduzione e collegarlo al resto del workflow nella maniera adeguata (questo significa, in pratica, collegare per ogni *attività* i flussi verdi in output con quelli in input dell'*attività* successiva, collegare, se ce ne sono, i *link* in output coi vari **Blue Port In** e così via).

4.1.2 Schemi di traduzione per le attività semplici di BPEL4WS

Qui di seguito esamineremo ancora una volta ognuna delle *attività* BPEL4WS per mostrare in che modo verrà tradotta mediante il secondo traduttore.

4.1.2.1 Empty

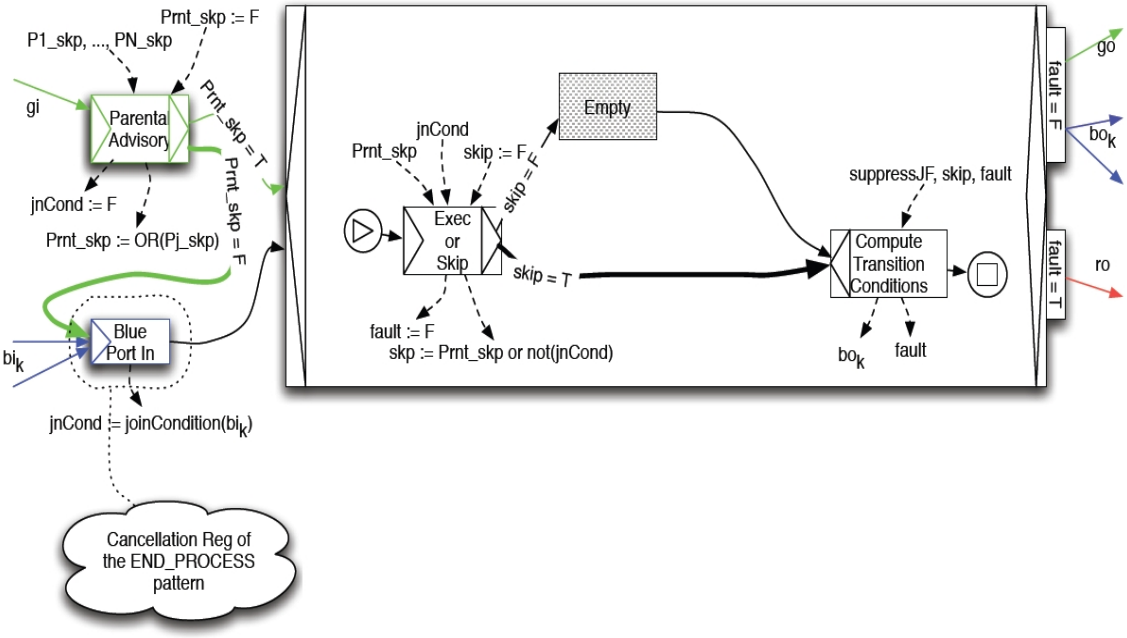


Figura 4.2: il pattern della Empty

Come è semplice notare l'*attività* **`empty`** non differisce molto dallo schema generale. Non genera implicitamente nessun tipo di fallimento così non è presente la parte che serve a determinare il successo o il fallimento dell'*attività*. La **Activity Specific Task** di questa *attività* è un task vuoto, che non fa nulla.

4.1.2.2 Receive

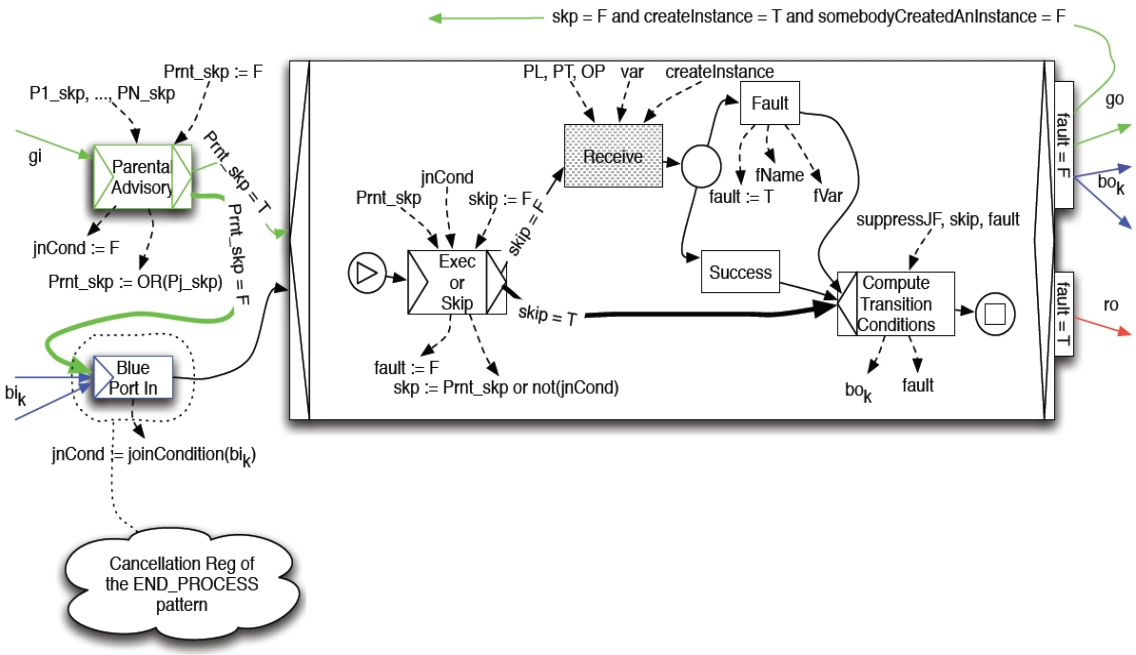


Figura 4.3: il pattern della Receive

La **`receive`** è l'*attività* che permette la ricezione di un messaggio dall'esterno. Come è possibile notare dalla figura il pattern di traduzione non differisce da quello dell'*attività* generica, ma c'è da notare che in caso non si verifichino degli errori e che l'*attività* non debba essere saltata, questa ha la possibilità di creare una istanza di processo se l'attributo **`createInstance`** ha il valore **`yes`**.

Questo, dal punto di vista del nostro workflow, ha solo il significato di attivare il gestore degli eventi, se nessuna altra *attività* aveva già creato l'istanza di processo precedentemente, per questo si usa la variabile **somebodyCreatedAnInstance** che

viene inizializzata, come vedremo più avanti, all'inizio del processo.

L'**Activity Specific Task** della **`receive`** simula la ricezione ottenendo in ingresso le variabili utili a stabilire una connessione con un servizio esterno (**`partnerLink`**, **`portType`** e **`operation`**).

Per quanto riguarda la possibilità di generare un errore nel task **Fault**, questo può succedere se si verificano errori nel messaggio ricevuto (come per esempio, quando non viene ricevuto un messaggio del tipo descritto nel documento WSDL e associato a quella **`receive`**). Il task **Fault** ha funzione analoga anche nell'*attività* **`invoke`** (dove il messaggio può essere sia ricevuto che inviato) e **`reply`** (dove il messaggio è solo inviato).

4.1.2.3 Reply

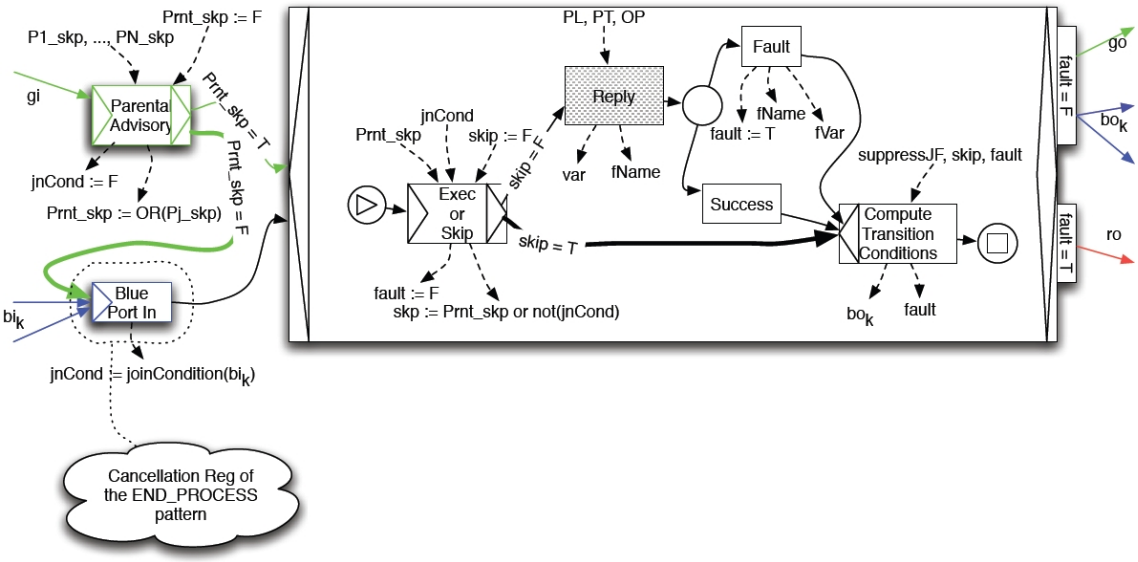


Figura 4.4: il pattern della Reply

L'*attività* **`reply`** è identica all'*attività* generica ad eccezione del suo **Activity Specific Task** che riceve in input i dati per la connessione a un servizio esterno e restituisce le eventuali informazioni d'errore, simulando in questo modo l'esecuzione della corrispondente *attività* BPEL4WS.

4.1.2.4 Wait

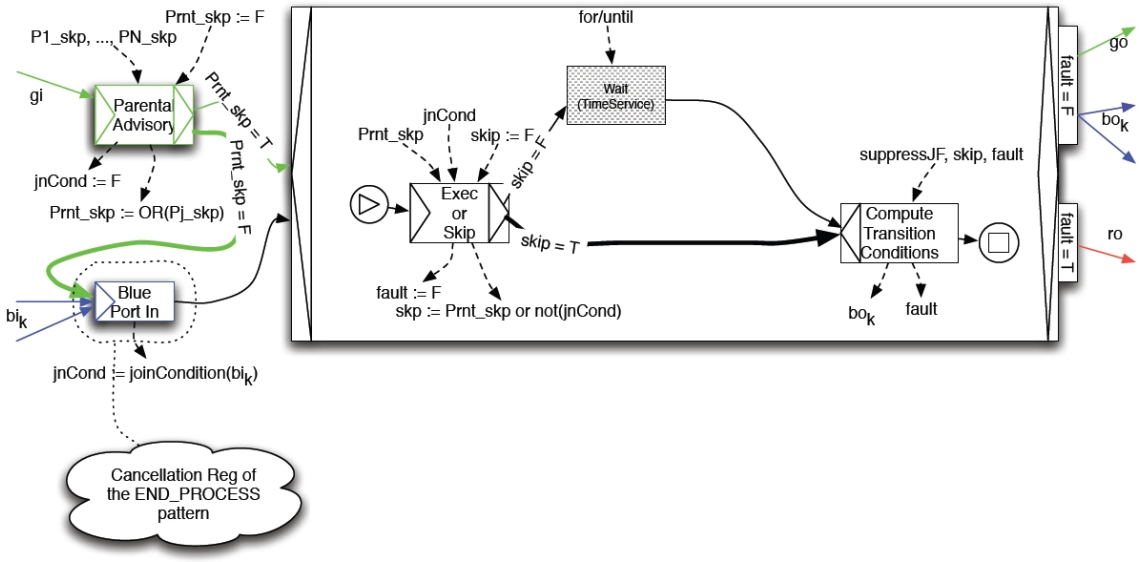


Figura 4.5: il pattern della Wait

La **`wait`** è una *attività* molto semplice, infatti si può notare che è strutturata come una **`empty`**. L'unica differenza sta nell'**Activity Specific Task** che prende in input la durata dell'attesa. E' inoltre possibile, se si dispone della versione dell'engine che gira su web server, connettere il task specifico con il servizio Web **time service** per permettere anche la simulazione dell'attesa

4.1.2.5 Invoke

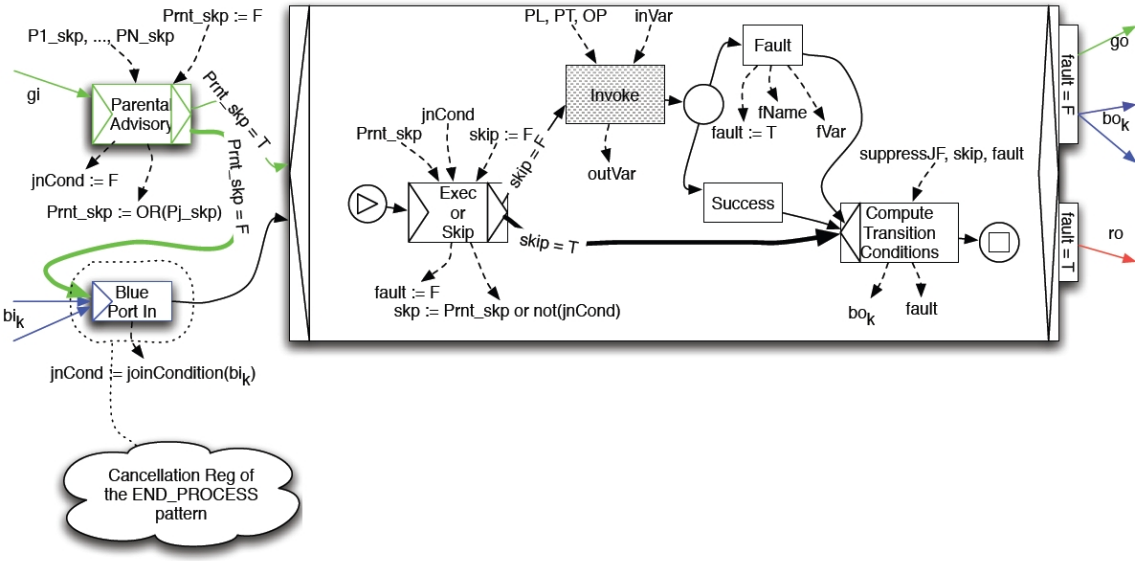


Figura 4.6: il pattern della Invoke

Per quanto riguarda la **`invoke`** non c'è da dire molto. Si comporta come una **`reply`** solo che la **Activity Specific Task** prende come input la variabile di input della stessa **`invoke`** e in output la variabile di output (sarà presente solo se l'invocazione è sincrona), come indicato dai valori **`inputVariable`** e **`outputVariable`** dell'attività (si veda il paragrafo *2.7.2* per maggiori dettagli).

4.1.2.6 Assign

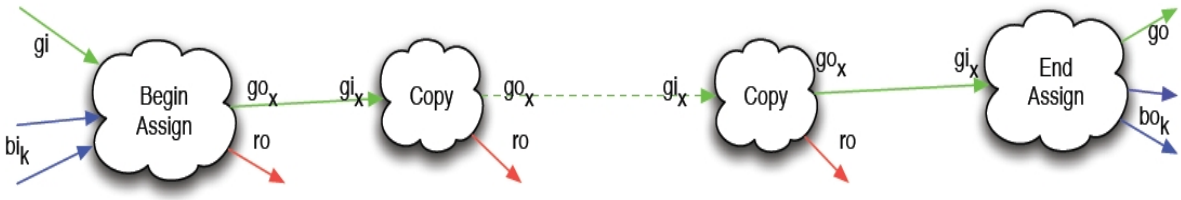


Figura 4.7: la Assign ad alto livello

La **`assign`** è la prima *attività* ad introdurre una struttura più complessa. Nella figura sopra ogni nuvola corrisponde ad un pattern di task come quelli visti precedentemente. Si può notare che per ogni elemento **`copy`** è presente un pattern, questo perché ognuno di questi può generare un errore. Visto che l'*attività* è divisa in due blocchi principali (**Begin Assign** e **End Assign**) che contengono gli altri, chiaramente la ricezione dei flussi blu che portano i *link* è lasciato alla **Begin Assign**, che quindi si occupa anche di segnalare eventuali errori di **`joinFault`**, mentre la **End Assign** si occuperà di collegare i *link* in uscita. Da notare che, una volta raggiunta la **End Assign**, avremo già stabilito se si sono verificati degli errori, e infatti nessun collegamento rosso esce da questo pattern, mentre tutti gli altri pattern, comprese le **`copy`**, possono generare errori.

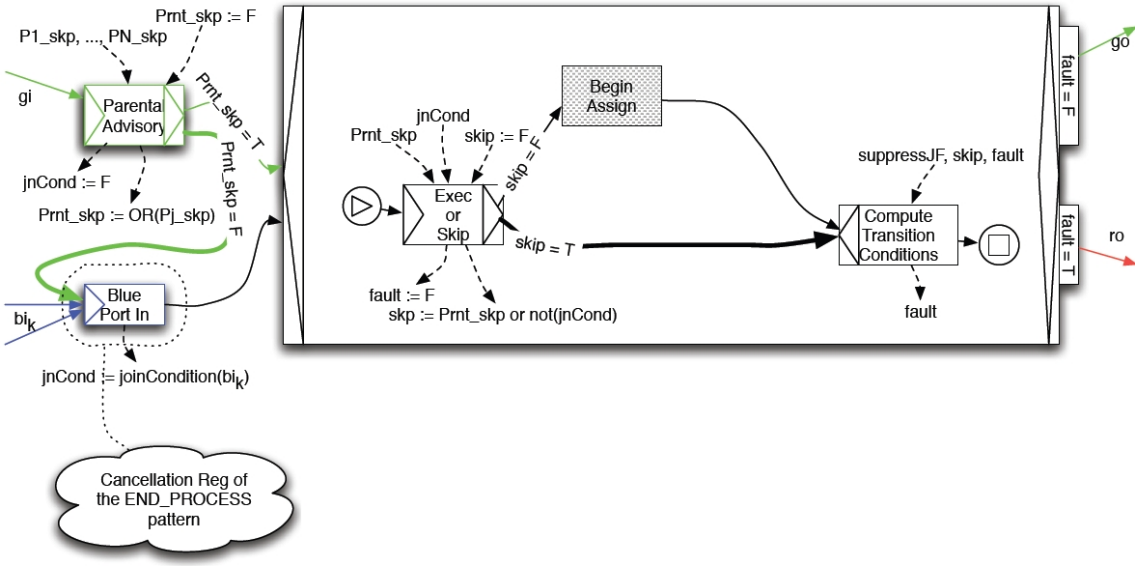


Figura 4.8: il pattern della Begin Assign

Come si può notare il pattern di traduzione della **Begin Assign** non comporta niente di particolare. Essendo un inizio di *attività* non può generare errori se non quello già menzionato di **`joinFault`**. Da notare che il pattern **Begin Assign** comporta un ulteriore livello di annidamento nei confronti dei pattern di copiatura, quindi il valore della sua variabile **skip** verrà passata ai pattern **Copy** come valore di **P1_skp, … , PN_skp**. Questo permetterà di saltare le **Copy** qualora l'intera **`assign`** debba essere saltata.

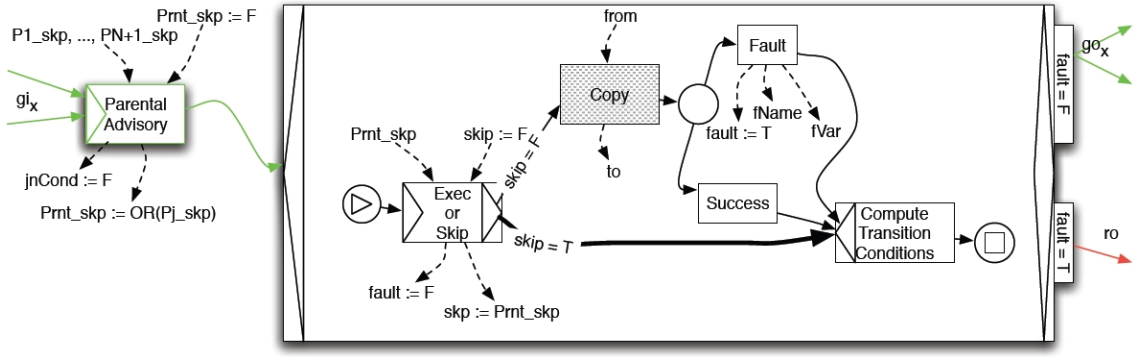


Figura 4.9: il pattern della Copy

La **copy** come già detto non necessita di un **Blue Port In** in quanto non rappresenta una *attività* BPEL4WS a se stante e quindi non può essere destinazione di *link*. Per il valore di **fName** in caso di fallimento, questo può essere solo un errore di assegnamento. Per il resto la **Activity Specific Task** prende un valore da **from** e lo copia in **to**. Ovviamente queste due variabili devono essere del tipo definito da BPEL4WS, come spiegato nel paragrafo *2.7.3*.

Da notare che il task **Exec or Skip** in questo caso non ha bisogno della variabile **jnCond**: questo perché tale variabile, che contiene il valore booleano della **joinCondition**, viene calcolato e interpretato solo nel pattern che simula l'inizio della **assign**.

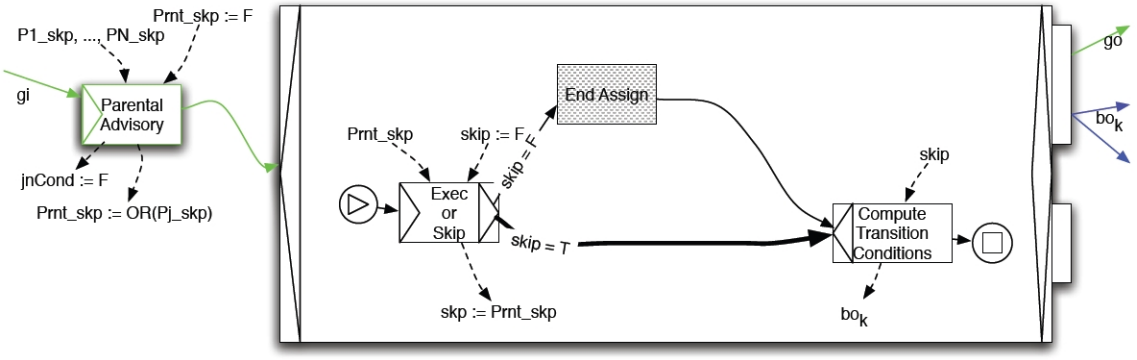


Figura 4.10: il pattern della End Assign

La **End Assign** si occupa praticamente solo di gestire i *link* in uscita. Non può essere destinazione di *link* in quanto non costituisce una *attività* BPEL4WS e non può generare

errori. Anche qui è da notare la mancanza della variabile **jnCond**, omessa anche in questo caso in virtù del fatto che il calcolo della **`joinCondition`** è lasciato al pattern iniziale.

4.1.2.7 Throw

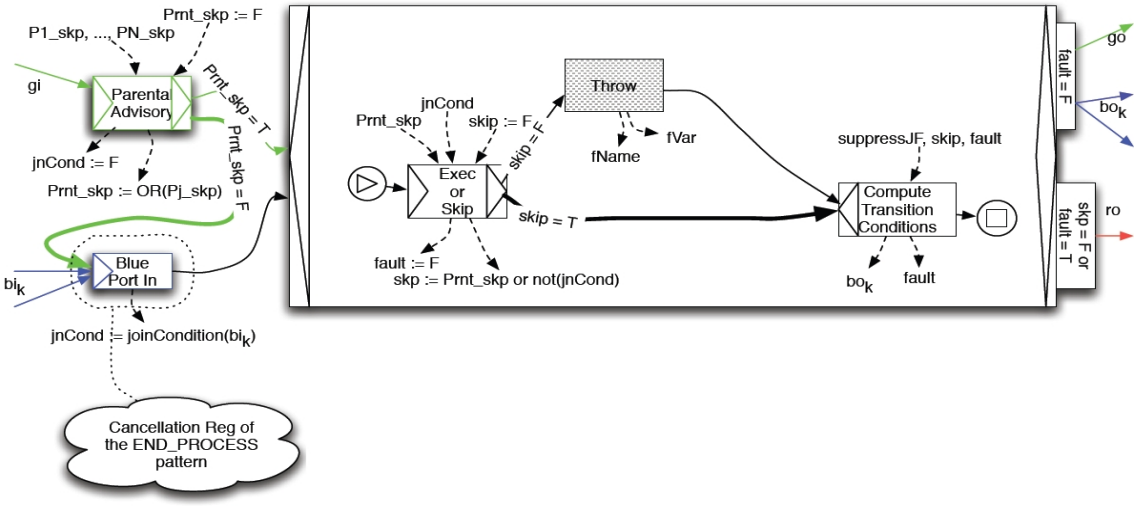


Figura 4.11: il pattern della Throw

La **`throw`** deve soltanto inviare il flusso al gestore dei fallimenti. Notare che tale flusso può essere inviato sia perché l'*attività* è stata eseguita, ma anche nel caso che venga saltata può essere generato un errore di **`joinFault`**. L'*attività* genera quindi come comportamento standard un errore, ed è quindi inutile la parte di pattern che serve a decretare il successo o il fallimento dell'esecuzione.

4.1.2.8 Compensate

La **`compensate`** ha una struttura particolare in quanto deve essere collegata al gestore della *compensazione* a cui si riferisce, e può invocarlo solo se lo **`scope`** in cui questo gestore è dichiarato è terminato con successo (poiché prima il gestore non risulta installato, come spiegato nel paragrafo *2.5.5*).

Si tenga presente che nella traduzione compiuta non si prenderà in esame la *compensazione* di default (che si ha nel caso in cui l'*attività* **`compensate`** venga invocata senza campo **`scope`**), poiché questa prevede la *compensazione* degli **`scope`** interni in ordine inverso a come questi vengono completati, e quindi si genera il problema di avere un ordinamento in caso di *attività* **`flow`**, che porterebbe questa attività a venire ulteriormente complicata, in quanto necessiterebbe di una pila in cui mantenere i dati sul completamento. Inoltre si avrebbe un ulteriore problema in presenza di una *attività* **`while`** interna allo **`scope`** da compensare perché se al suo interno vi fosse un altro **`scope`** ogni istanza

Lo schema ad alto livello della **`compensate`** risulta quindi il seguente:

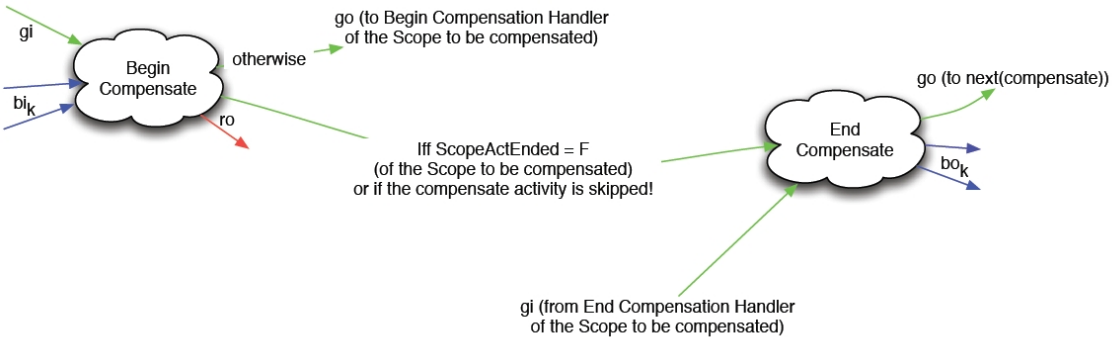


Figura 4.12: la Compensate ad alto livello

La **`compensate`** diventa quindi una sorta di contenitore per una **`compensationHandler`**. Se però lo **`scope`** non ha ancora terminato la propria esecuzione la **`compensate`** salta la propria esecuzione proprio come se dovesse saltarla a causa dei *link*. Questa è una rappresentazione del comportamento reale di BPEL4WS che, come spiegato nel paragrafo *2.5.5*, in questi casi prevede l'esecuzione di una *attività* **`empty`** al posto del gestore.

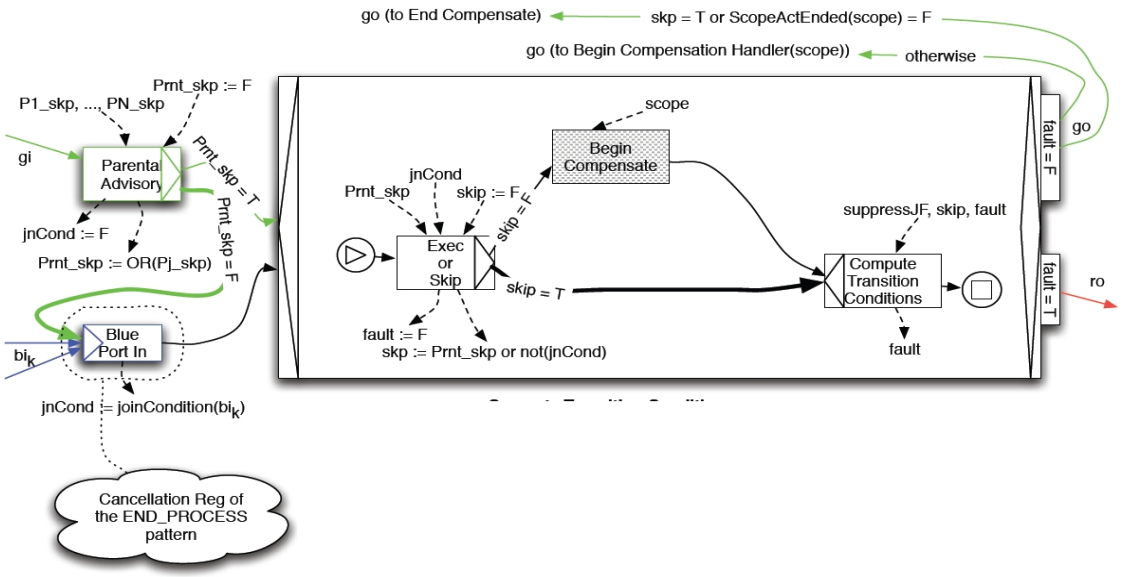


Figura 4.13: la Begin Compensate

La **Begin Compensate** si comporta quindi come previsto accettando i **blue inputs** e calcolandone nella **Blue Port In** la **`joinCondition`** che poi verrà passata al **Main Task**. La **Activity Specific Task** riceve in input il nome dello **`scope`** da compensare solo per mantenere il riferimento a tale **`scope`** dopo la traduzione.

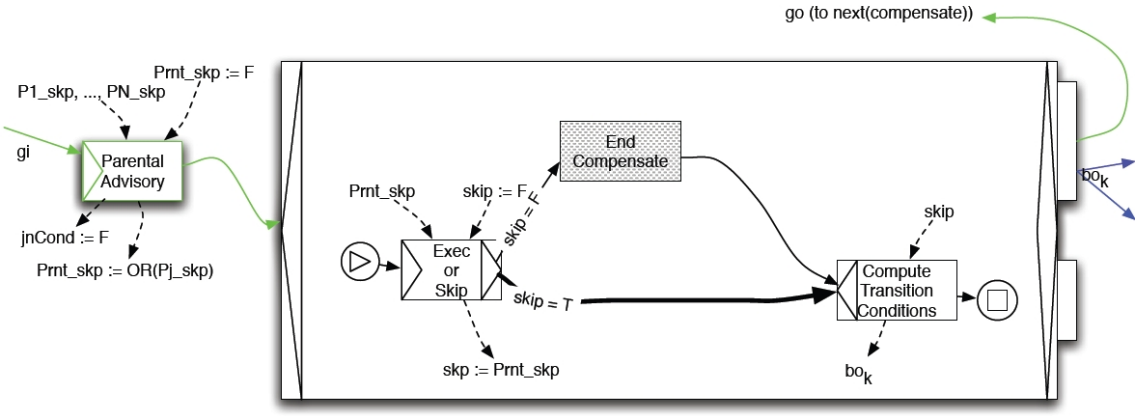


Figura 4.14: la End Compensate

La **End Compensate** non ha da svolgere nessun compito a parte calcolare il valore booleano dei *link* della **`compensate`** stessa. Essendo solo la parte finale dell'*attività* non può generare nessun tipo di fallimento e non riceve *link* in ingresso, e la sua **Activity**

Specific Task non deve svolgere nessun compito. Da notare anche in questo caso la mancanza della variabile **jnCond**, che non viene utilizzata in quanto il compito di calcolare il valore della **`joinCondition`** viene lasciato alla **Begin Compensate**.

4.1.2.9 Terminate

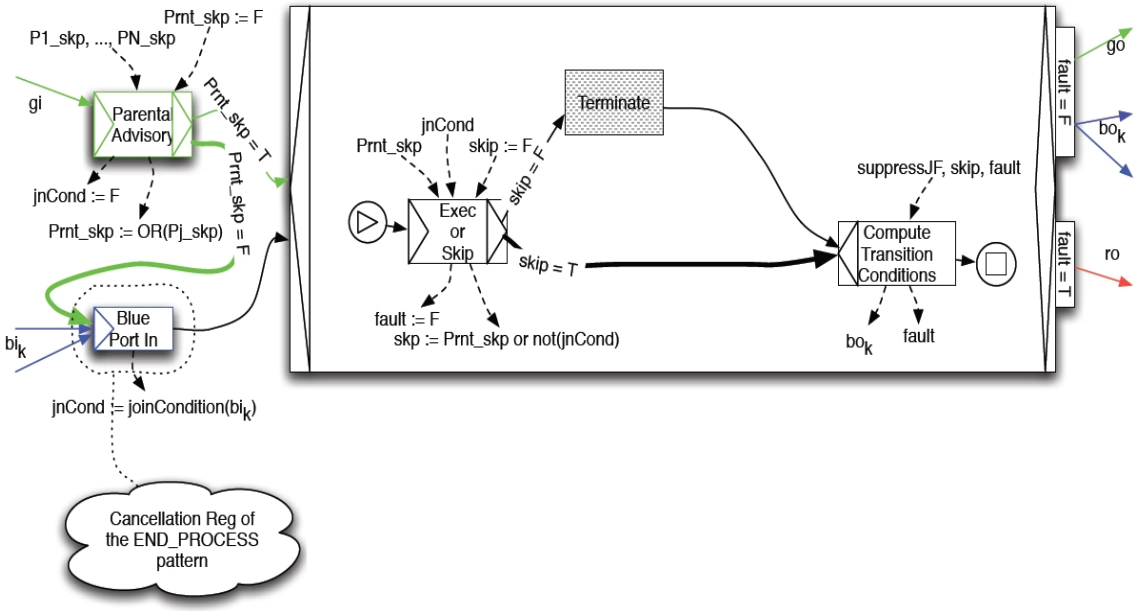


Figura 4.15: il pattern della Terminate

La **`terminate`** deve far terminare l'intero processo. Per questo motivo il suo compito, in YAWL, si traduce nel mandare il flusso al pattern che rappresenta la fine del processo e di bloccare l'esecuzione di qualsiasi altra attività, cancellando tutti i token di tutte le attività, ad eccezione dell'attività di fine processo. Considerato che della cancellazione dei token si occupa il pattern di fine processo, la **`terminate`** svolge solo il compito di mandare il token all'**End Process**.

Per quanto riguarda il resto, il pattern di per sé non ha molte particolarità. Non potendo generare errori, se non il solito di **`joinFault`**, la traduzione risulta identica a quella della **`empty`**.

4.1.3 Schemi di traduzione per le attività composite di BPEL4-WS

Per le *attività* composite si usano generalmente almeno due pattern che indicano l'inizio e la fine dell'*attività* nella traduzione YAWL.

Vedremo che per le *attività* composte è sempre presente una struttura di questo genere dove **Begin** logicamente corrisponde all'inizio dell'esecuzione dell'*attività*. **End** logicamente corrisponde alla fine dell'esecuzione dell'*attività*. Di conseguenza, **Begin** manda un token verde a ogni figlio che si debba eseguire all'inizio e rispettivamente, **End** aspetta un token verde da ogni figlio che si debba eseguire alla fine. Questo discorso spiega anche perché **Begin** è il target dei *link* (ovvero la destinazione dei **blue input**) ed **End** e la sorgente (ovvero il mittente dei **blue output**).

Per quanto riguarda le *attività* interne il valore di **P1_skp, … , PN_skp** nel pattern sarà quello della variabile **skip** del pattern che li racchiude, per fare sì che il salto causato da una **`joinCondition`** che non risulti essere verificata nell'*attività* composita si ripercuota anche in tutte le *attività* interne, come richiesto dalla specifica di BPEL4WS. Si noti che in questo caso tutte le sorgenti di *link* interne all'*attività* composita dovranno applicare la *dead-path elimination* come è stato spiegato nel paragrafo *2.6.1*.

4.1.3.1 Sequence

Lo schema ad alto livello della **sequence** è il seguente:

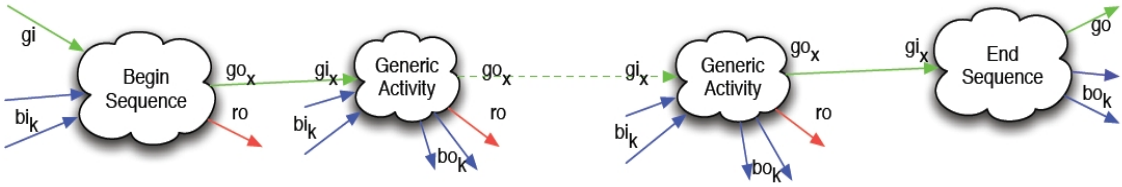


Figura 4.16: la Sequence ad alto livello

La **sequence** introduce un altro livello di annidamento quindi, come detto prima, le *attività* interne ottengono in input il valore di **skip** della **sequence** stessa, in modo da

essere saltate se questa viene saltata.

Ogni nuvola etichettata con **Generic Activity** si riferisce al pattern che descrive una *attività* figlia, che, nel caso delle *attività* composite, può essere chiaramente formata da più pattern.

Il pattern **Begin Sequence** non differisce molto dalla **Begin Compensate**. Noteremo che tutti i pattern di inizio *attività* sono molto simili tra loro, in quanto il loro compito è quello di valutare la **`joinCondition`**, decidere un eventuale salto oppure far svolgere altre *attività* o pattern che eseguano altri compiti.

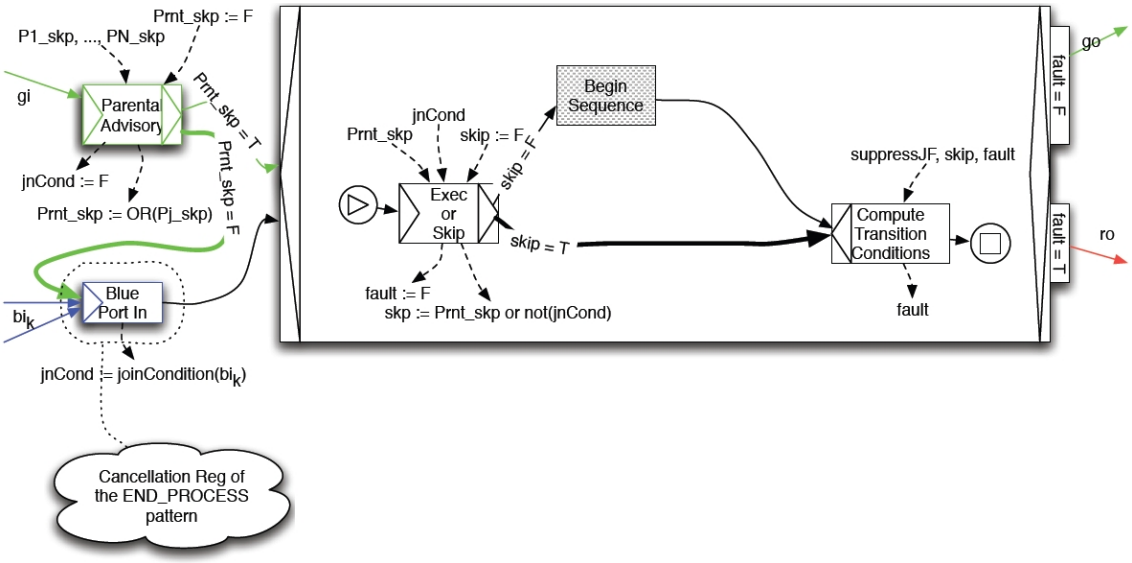


Figura 4.17: il pattern Begin Sequence

Anche per quanto riguarda i pattern di fine *attività* le analogie sono molte: il loro compito è soltanto valutare i *link* in output. Quindi di solito non generano errori, non possono avere il task **Blue Port In** e non valutano la **`joinCondition`**.

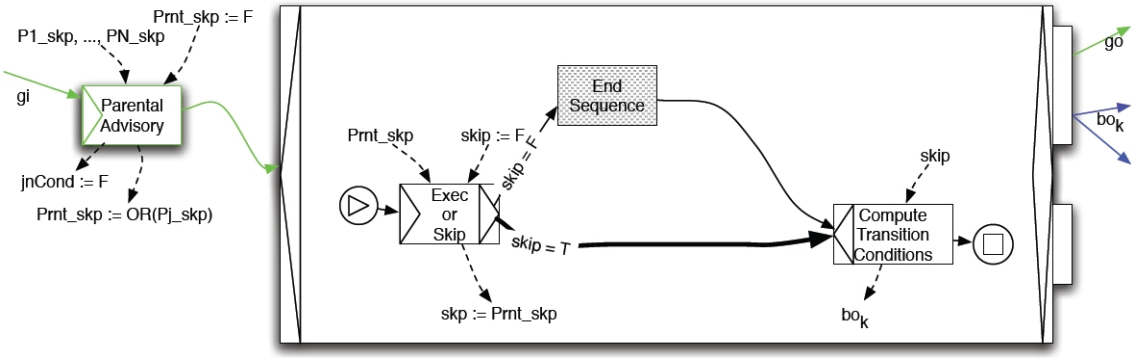


Figura 4.18: il pattern End Sequence

4.1.3.2 Flow

Il funzionamento della **`flow`** ad alto livello è identico a quello della prima versione del traduttore: la **Begin Flow** genera un nuovo livello di annidamento indispensabile per saltare tutte le *attività* interne qualora la **`flow`** dovesse essere saltata. Le *attività* interne sono collegate alla **Begin Flow** tramite un AND-split, in modo da essere mandate in esecuzione tutte contemporaneamente, mentre la **End Flow** aspetta, per mezzo dell'AND-join, che tutte le attività terminino.

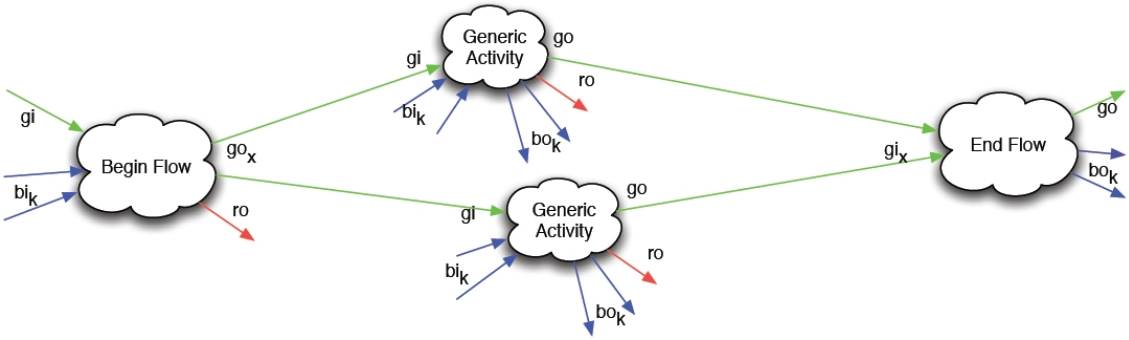


Figura 4.19: la Flow ad alto livello

I pattern **Begin Flow** e **End Flow** non hanno nessuna differenza rispettivamente con la **Begin Sequence** e la **End Sequence** in quanto il compito svolto da questi due pattern è esattamente lo stesso: il primo riceve i link e calcola se c'è un **`joinFault`**, il secondo

si occupa delle connessioni dei *link* di cui l'*attività* è sorgente.

4.1.3.3 Switch

L'*attività* **`switch`** viene qui tradotta in maniera tale che ad alto livello risulta analoga alla **`sequence`** ed esegue ogni **`case`** in sequenza, invece che in parallelo.

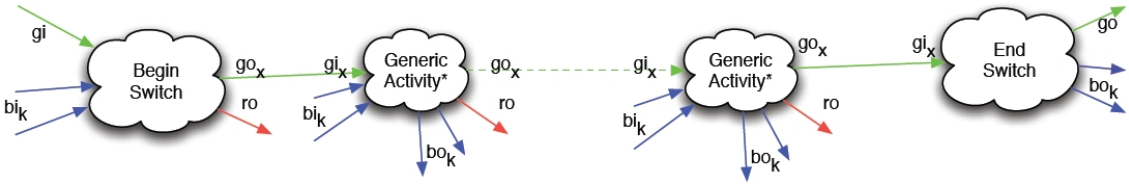


Figura 4.20: la Switch ad alto livello

Come per le due *attività* appena considerate, anche la **`switch`** introduce un nuovo livello di annidamento che viene considerato solo dai pattern che rappresentano i vari **`case`**.

Le **Generic Activity** che rappresentano i vari **`case`** sono ordinate con lo stesso ordine che assumono all'interno del file BPEL4WS. Non c'è bisogno di introdurre pattern aggiuntivi per gestire la scelta del **`case`** da eseguire: il **Parental Advisory** delle *attività* contenute all'interno dell'elemento **`case`** verrà modificato affinché determini se quel ramo deve essere scelto.

Nel caso in cui mancasse il ramo **`otherwise`**, secondo quanto detto nel paragrafo *2.8.3*, verrà introdotto un ramo **`otherwise`** immesso subito prima della **End Switch** e creato con una *attività* **`empty`** al suo interno.

Per la **Begin Switch** e la **End Switch** si rimanda alla **Begin Sequence** e alla **End Sequence** visto che i due pattern sono identici. Per quanto riguarda invece le *attività* all'interno dei **`case`** esse vengono modificate e collegate in sequenza come nella figura sopra. Le modifiche sono riportate nel seguente schema:

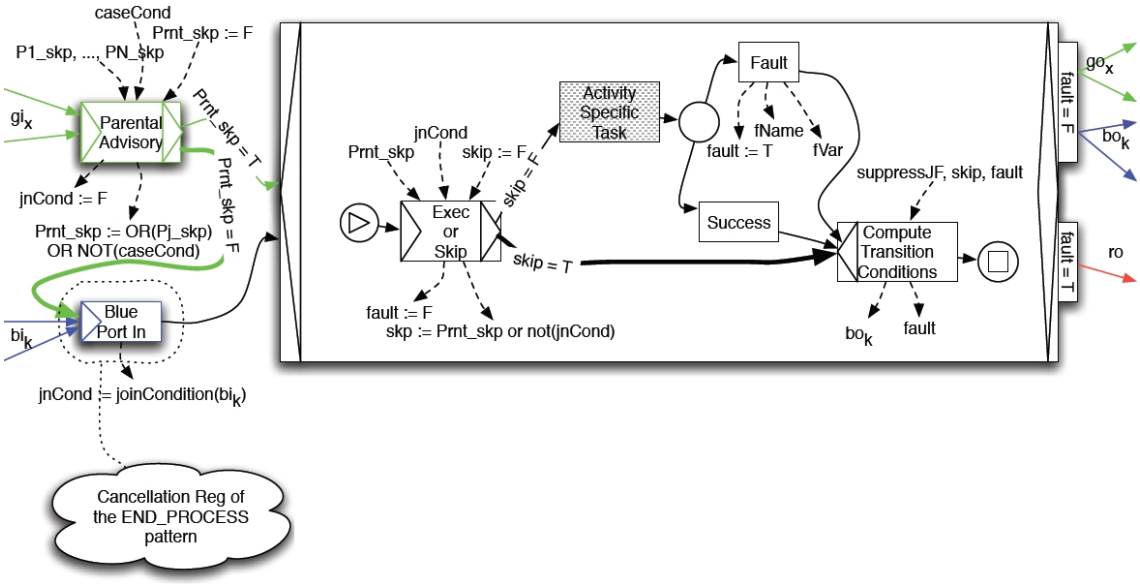


Figura 4.21: le modifiche al pattern generico per la Case

Nel caso di una *attività* composita le modifiche da fare riguarderanno il pattern iniziale. Come si nota dalla figura l'unica modifica da apportare è, appunto, nel **Parental Advisory**, che deve prendere in input la condizione dello specifico **`case`** (nel caso di **`otherwise`** tale condizione sarà **true()**, ovvero la funzione XPath che restituisce sempre il valore vero) e che calcola il suo **Prnt_skp** basandosi anche su questa condizione.

Per tutto il resto il pattern rimane invariato.

4.1.3.4 While

Nella **`while`** si ripete l'*attività* interna finché la condizione non diventa falsa. Nella traduzione si utilizza lo schema seguente:

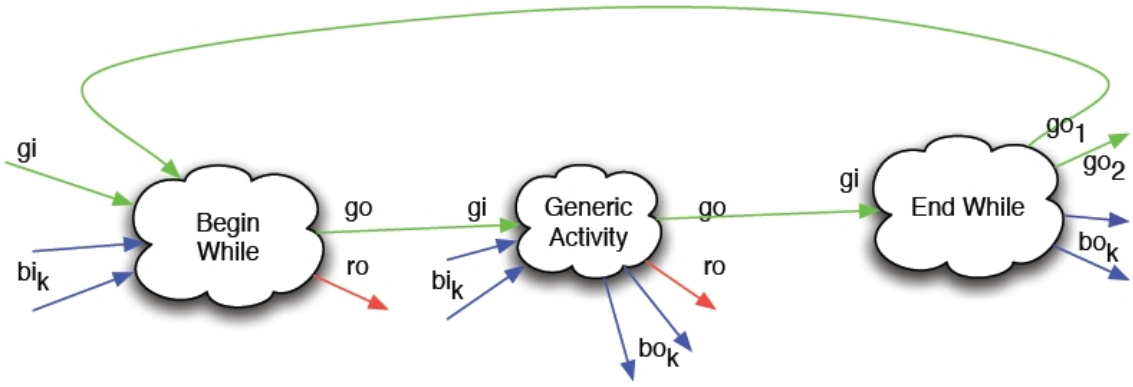


Figura 4.22: la While ad alto livello

C'è una cosa da tenere in considerazione però in questo tipo di *attività*: i *link* delle attività interne alla **`while`** non possono superare i confini della **`while`** stessa.

Per quanto riguarda il salto dell'*attività* interna, anche la **`while`** introduce un ulteriore livello di annidamento, quindi la **Prnt_skp** dell'*attività* che costituisce il corpo della **`while`** sarà la **skip** della stessa **`while`**.

Da notare che nella traduzione proposta sia la **Begin While** che la **End While** eseguono il controllo sulla condizione per le ripetizioni dell'*attività*. Questo perché nella **Begin While** si valuta se dover eseguire l'*attività*, mentre nella **End While** si valuta se è necessario fare un altro ciclo. Si poteva fare in modo di evitare questo doppio test, ma in quel caso si sarebbe comunque dovuto sempre compiere un giro a vuoto di tutto il pattern e si sarebbe eseguita una *dead-path elimination* superflua che doveva essere controllata in qualche modo, altrimenti avrebbe potuto portare ad una esecuzione errata del processo.

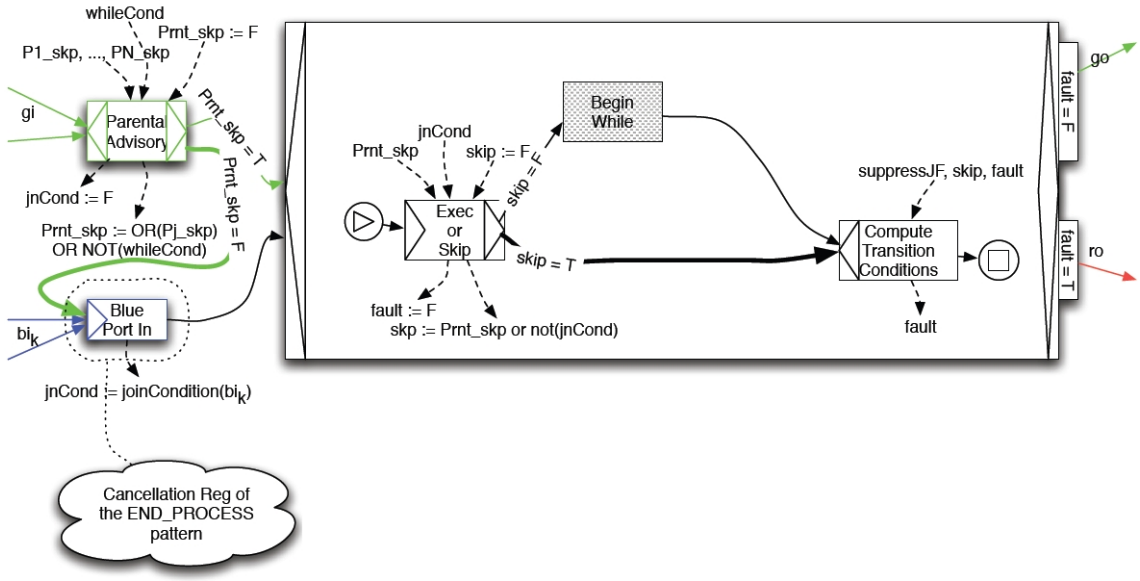


Figura 4.23: il pattern Begin While

Come si può notare la **Begin While** non differisce molto dai pattern iniziali delle altre *attività* composite. Oltre a non generare fallimenti a parte il **`joinFault`**, l'unica differenza si può notare nel task **Parental Advisory** che riceve in input la condizione del **`while`** e calcola il proprio **Prnt_Skp** in funzione di quella.

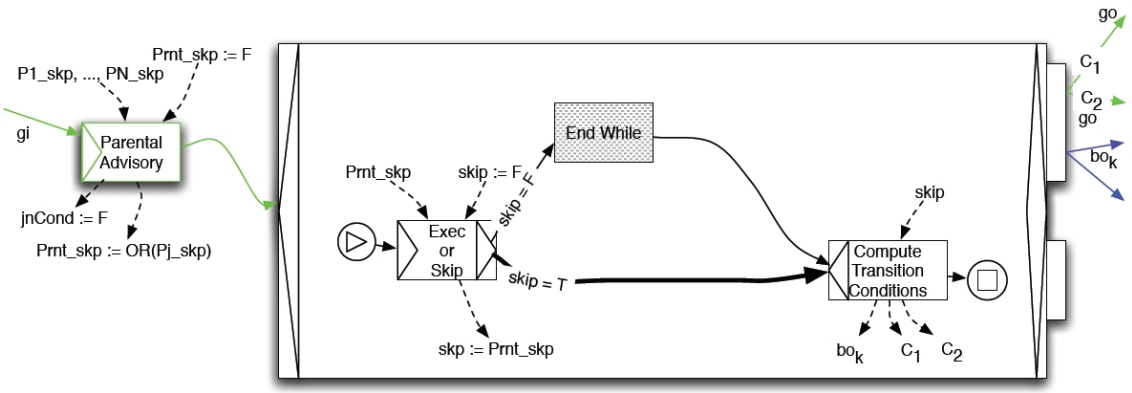


Figura 4.24: il pattern End While

La **End While** calcola la **skip** al solito modo ma per definire quale dei due flussi seguire usa questa informazione in unione con la condizione del **`while`**. Questo perché comunque i *link* sono condizionati solo dalla **skip** e non dalla ripetizione dell'*attività*.

Nella figura **C1** rappresenta il flusso che riporta all'inizio del **`while`**, che viene seguito se è falsa la variabile **skip** e se risulta vera la condizione del **`while`**, mentre **C2** quello che prosegue, e viene seguito quando non si segue **C1**.

4.1.3.5 Pick

Simulare una **`pick`** in un workflow YAWL non è una cosa banalissima. Lo schema di traduzione proposto, anche se a prima vista può sembrare un po' complicato, riesce a coprire tutte le possibilità che si possono generare.

La **`pick`** attende che o un messaggio venga ricevuto o un allarme venga eseguito. Quindi la simulazione deve scegliere quale di queste due possibilità si verifica per prima ed eseguirne l'*attività* correlata. Vista ad alto livello, la **`pick`** deve eseguire in parallelo tutte le *attività* sue figlie, ma solo chi è stata selezionata verrà eseguita realmente, le altre saranno saltate.

Lo schema di traduzione ad alto livello è il seguente:

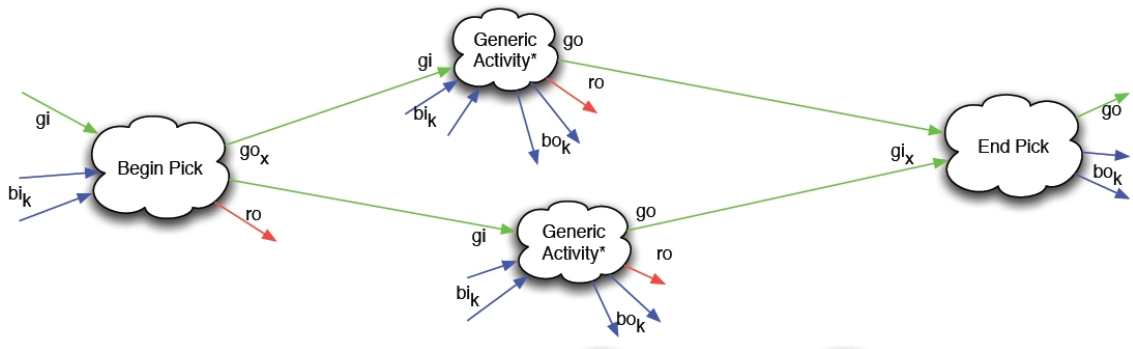


Figura 4.25: la Pick ad alto livello

La scelta di quale *attività* eseguire viene compiuta nella **Begin Pick**, considerato che le ricezioni di messaggi devono essere simulate e le attese degli allarmi sono simulate anch'esse (in questo caso al limite si potrebbe sfruttare il Time Service della versione su web server dell'engine). In pratica le **`onMessage`** si comportano approssimativamente come delle **`receive`** e le **`onAlarm`** come delle **`wait`**.

Lo schema di traduzione della **Begin Pick** è il seguente:

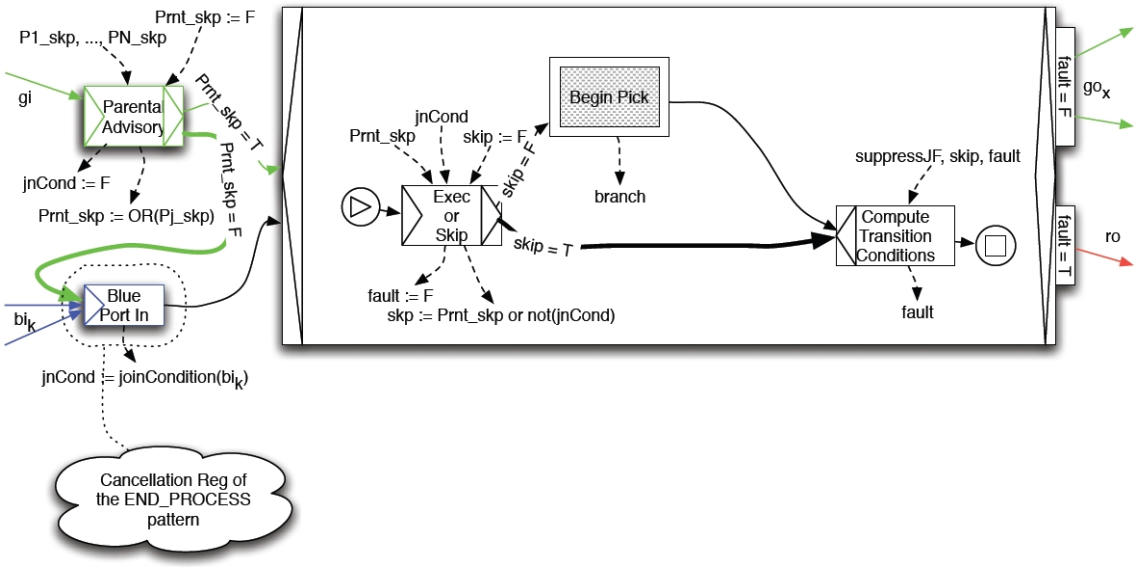


Figura 4.26: il pattern della Begin Pick

Questo pattern non differisce molto dagli altri pattern iniziali, se non nella **Activity Specific Task**. In questo caso abbiamo un task composito, la cui esecuzione dà in output una variabile, "**branch**", che contiene l'identificativo del ramo che deve essere selezionato per l'esecuzione.

Questa variabile sarà poi copiata nella rete principale e verrà presa come input da tutte le *attività* figlie della **`pick`**.

Ma prima di analizzare il funzionamento delle *attività* interne è necessario esaminare come sia formato il task composito **Begin Pick**. La figura seguente ne mostra la sottorete, in maniera generica:

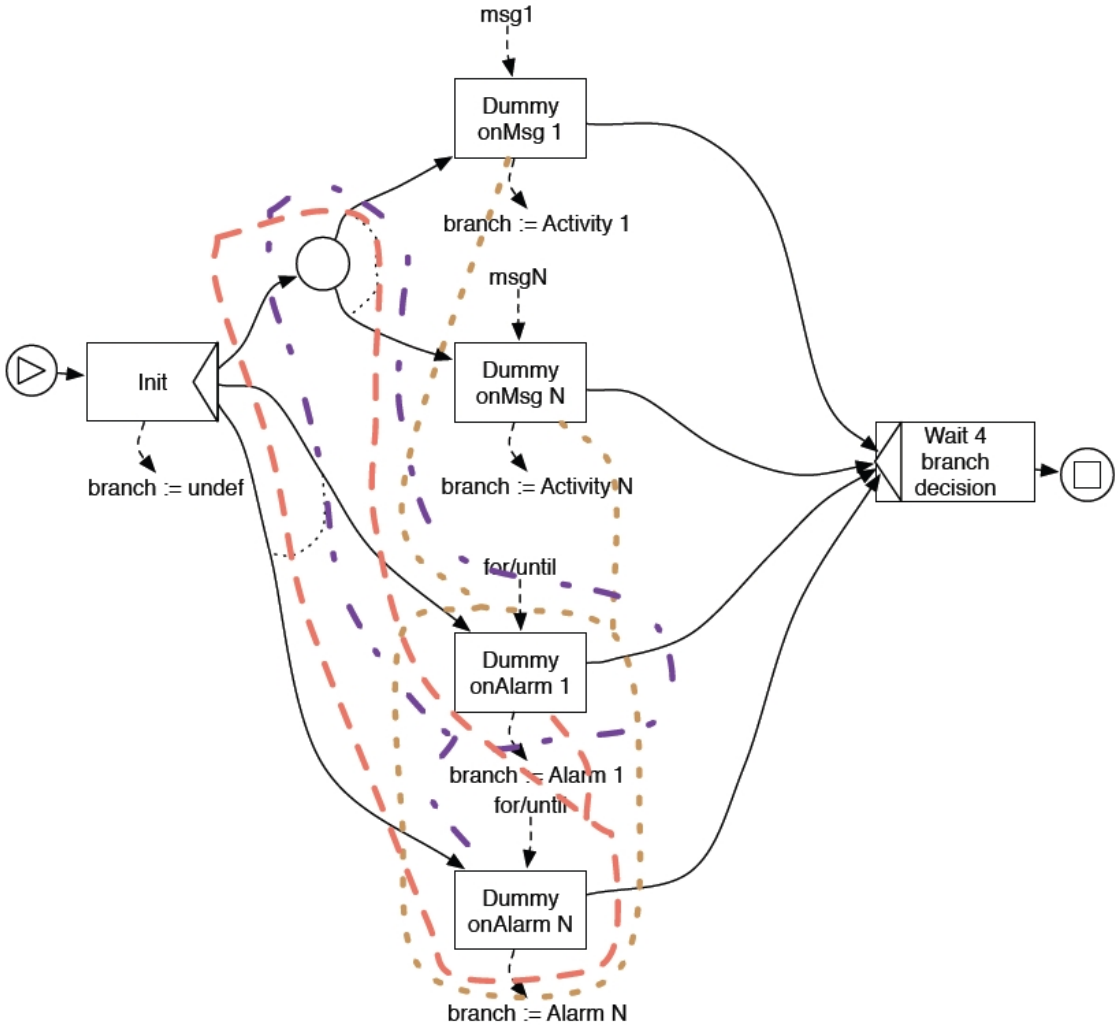


Figura 4.27: la sottorete della Begin Pick

Il funzionamento non è particolarmente complesso: la **Init** manda in esecuzione tutti i **Dummy onAlarm**, che, ricordiamo, possono eseguire una chiamata al Time Service, e la `condition` (che permette una *scelta differita* del flusso da seguire, come spiegato nel pattern 16 del paragrafo *3.2*).

Tutti i **Dummy onAlarm** hanno nel loro *cancellation set* gli altri **Dummy onAlarm** e la `condition`, in modo da cancellare l'esecuzione di qualsiasi altro task interno a questa sottorete qualora venissero eseguiti essi stessi.

D'altra parte solo uno dei **Dummy onMsg**, che rappresentano l'attesa dei messaggi

ricevuti, può essere mandato in esecuzione dopo la `condition`. Quindi nel loro *cancellation set* ognuno di loro avrà tutti i **Dummy onAlarm**.

Tutti questi task svolgono un compito solo: quello di segnalare tramite la variabile "**branch**" il proprio identificativo, che corrisponde a quello del ramo che verrà in seguito scelto per continuare l'esecuzione della `pick`.

Solo il flusso che partirà dal task eseguito per primo tra i **Dummy** raggiungerà il task finale **Wait 4 Branch Decision** e questo farà terminare l'esecuzione della sottorete. La **Begin Pick** intesa come **Activity Specific Task** avrà ora la sua variabile inizializzata con l'identificativo giusto e si potrà continuare nell'esecuzione del workflow.

A questo punto verranno mandati in esecuzione i task che rappresentano le *attività* interne alle `onMessage` e `onAlarm`, opportunamente modificate affinché non vengano eseguite se non sono state selezionate nella sottorete appena descritta. Le modifiche da compiere per un pattern che rappresenta una *attività* all'interno di una `onAlarm` riguardano solo il task **Parental Advisory**, che viene modificato come segue:

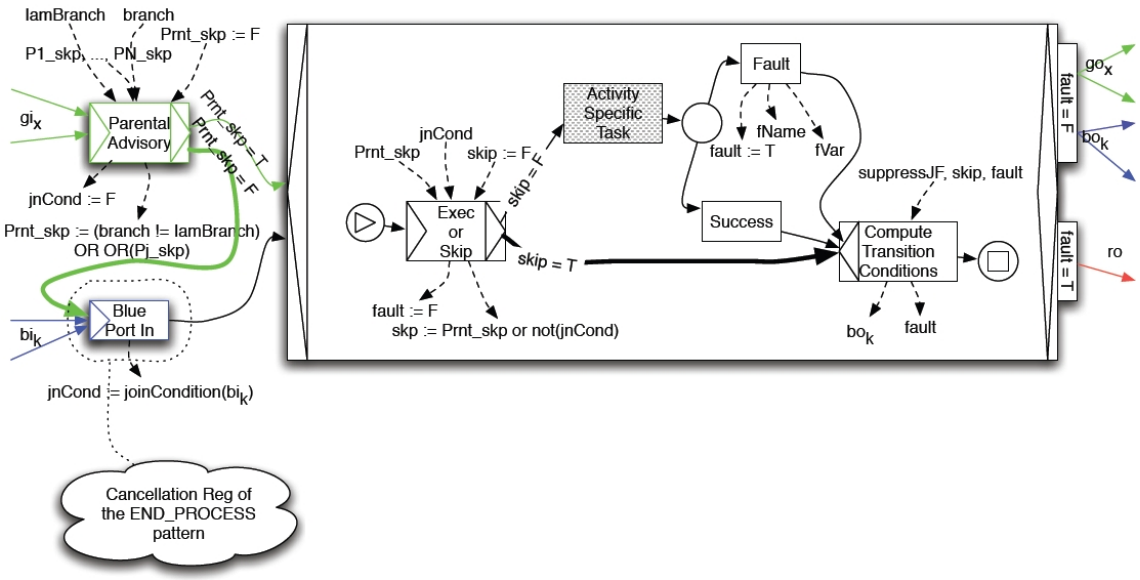


Figura 4.28: il pattern generico modificato dalla onAlarm

IamBranch è una variabile che contiene l'identificativo di questo pattern. Per venire eseguito tale identificativo deve corrispondere a quello scelto nella sottorete della **Begin**

Pick e, ovviamente, l'attività non deve essere saltata. Per il resto il pattern risulta essere identico a quello di una *attività* generica.

L'attività nel caso della **`onMessage`** risulta modificata allo stesso modo, solo viene gestita anche la **`createInstance`**.

La **`pick`** termina con un pattern finale identico a quello delle altre *attività* composite che gestisce i *link* di cui questa *attività* è sorgente e non può generare fallimenti.

4.1.3.6 Scope e gli handler

Questa *attività* è la più complessa di tutte. Per spiegarla verrà da prima introdotta la sua rappresentazione ad alto livello, che poi verrà commentata parte per parte.

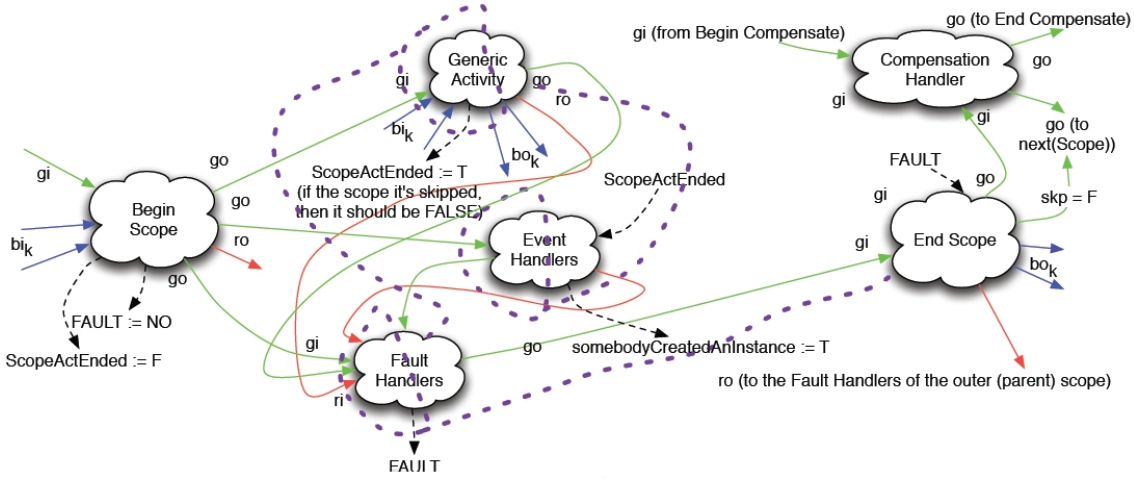


Figura 4.29: la Scope ad alto livello

La **Begin Scope** ha il compito, oltre che di ricevere i *link* in input e calcolare se si genera un eventuale **`joinFault`**, di inizializzare due variabili che devono essere specifiche di questo pattern, chiamate **FAULT** (che mantiene il valore di eventuali fallimenti generati nella **`scope`**) e **ScopeActEnded** (valore booleano che indica se questa **`scope`** ha terminato con successo la propria esecuzione).

Fatto ciò viene mandato in esecuzione in parallelo sia il pattern dell'*attività* interna alla **`scope`**, sia il gestore degli eventi che quello dei fallimenti. Per poter essere

completato il pattern deve passare proprio dal **faultHandlers** sia esso eseguito o saltato.

L'*attività* interna della **scope** deve mandare il proprio token al **faultHandlers**. Questo vuol dire che nel caso in cui l'*attività* non sia saltata, i due pattern verranno eseguiti in sequenza e il **faultHandlers** verrà saltato. Se invece l'intero pattern non deve essere eseguito perchè tutta l'*attività* deve essere saltata (ricordiamo che anche la **scope** introduce un ulteriore livello di annidamento), **Generic Activity** e **faultHandlers** saranno saltati in parallelo, ma il token dalla **Generic Activity** arriverà nel task **Parental Advisory** del pattern di inizio **faultHandlers**, verrà propagato nel task **RG2 Port In** e qui rimarrà bloccato in attesa di un token da **RG1 Port In** e sarà compito della **End Scope** quello di cancellarlo (si veda più avanti la struttura della **Begin Fault Handlers**).

Come mostra la figura il **Begin Fault Handlers** cancella tutti i token dell'*attività* interna alla **scope** ad eccezione di quelli dei **compensationHandler** delle **scope** annidate all'interno dell'attività generica, quando questo riceve un **red input**.

Per quanto riguarda l'**eventHandlers**, questo riceve un token solo se la **scope** non deve essere saltata, per evitare che le *attività* al suo interno generino una esecuzione di salti ciclica. In più i link non possono superare i confini dell'**eventHandlers** quindi non c'è bisogno della *dead-path elimination* al suo interno.

La variabile **somebodyCreatedAnInstance**, definita globale in quanto riguarda tutto il processo, indica se è stata creata una istanza di processo e quindi se deve essere eseguito o saltato il pattern dell'**eventHandlers**.

La **Begin Scope** svolge gli stessi compiti dei pattern iniziali delle *attività* strutturate con la sola differenza che se non viene saltato inizializza le variabili **FAULT** a **NO** e **ScopeActEnded** a **False**.

Anche la **End Scope** è un pattern identico a quelli finali delle altre *attività* composite. Solo che in questo caso i flussi in cui viene mandato un token possono essere due: quello che porta alla **Begin Compensation Handler** che viene inviato sempre, e quello all'*attività* successiva alla **scope**, che viene inviato solo se la **scope** non è stata saltata.

Il compito della **End Scope** differisce da quello dei generici pattern di fine *attività*

anche perché può mandare un **red output** nel caso in cui non sia stato possibile gestire un eventuale fallimento all'interno della **`scope`**. In questo caso bisogna propagare un token rosso al gestore della **`scope`** padre o del processo intero (come vedremo nel paragrafo *4.1.5*). In più ha il compito, se la **`scope`** è andato a buon fine, di mandare un token anche al **Begin Compensation Handlers** per 'attivarlo', ovvero per permettere a una eventuale **`compensate`** di mandarlo in esecuzione.

Lo schema seguente mostra il pattern del **`faultHandlers`** ad alto livello:

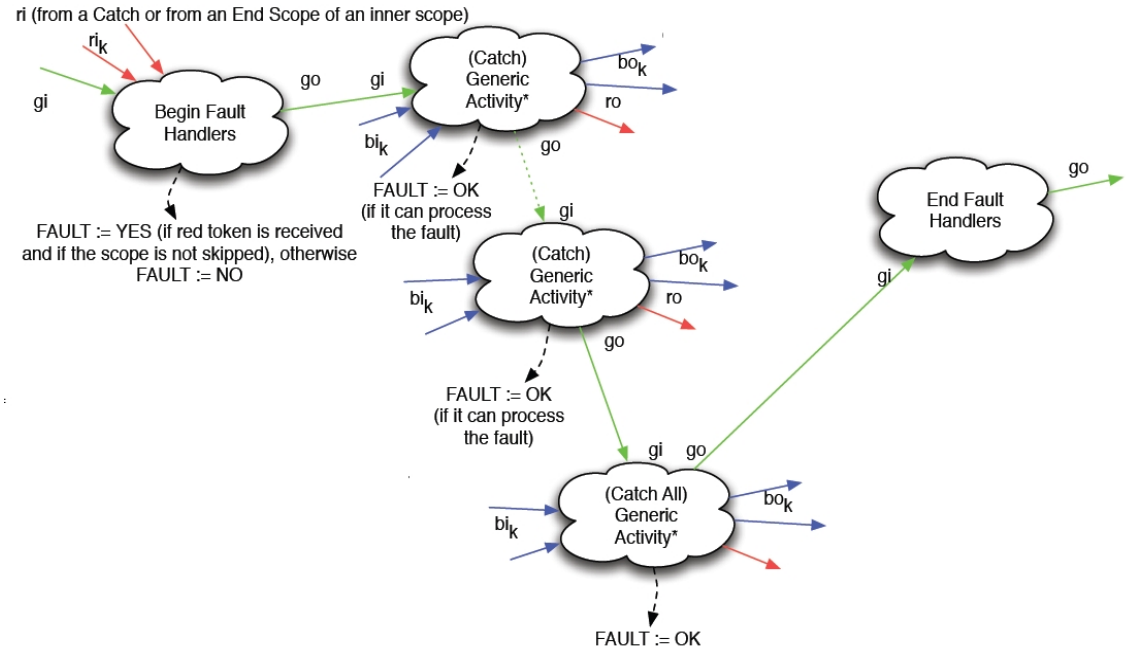


Figura 4.30: il Fault Handlers ad alto livello

Notiamo subito che se non è definito un **`faultHandlers`** per la **`scope`** il traduttore ne deve generare uno di default che sia composto solo dalla **Begin Fault Handlers** e dalla **End Fault Handlers** in modo da permettere alla **End Scope** di poter propagare eventuali fallimenti non gestiti.

Il funzionamento del **`faultHandlers`** ricorda molto i pattern dell'*attività* **`switch`**: il primo che può gestire il fallimento lo fa, segnala che la gestione c'è stata (**FAULT** assume il valore **OK**) e continua a propagare il flusso ai compagni, che però verranno

saltati. Se invece non c'è bisogno di gestione e il **`faultHandlers`** viene saltato il pattern **Begin Fault Handlers** lo segnala inizializzando a **NO** la variabile **FAULT** e tutto il pattern viene saltato.

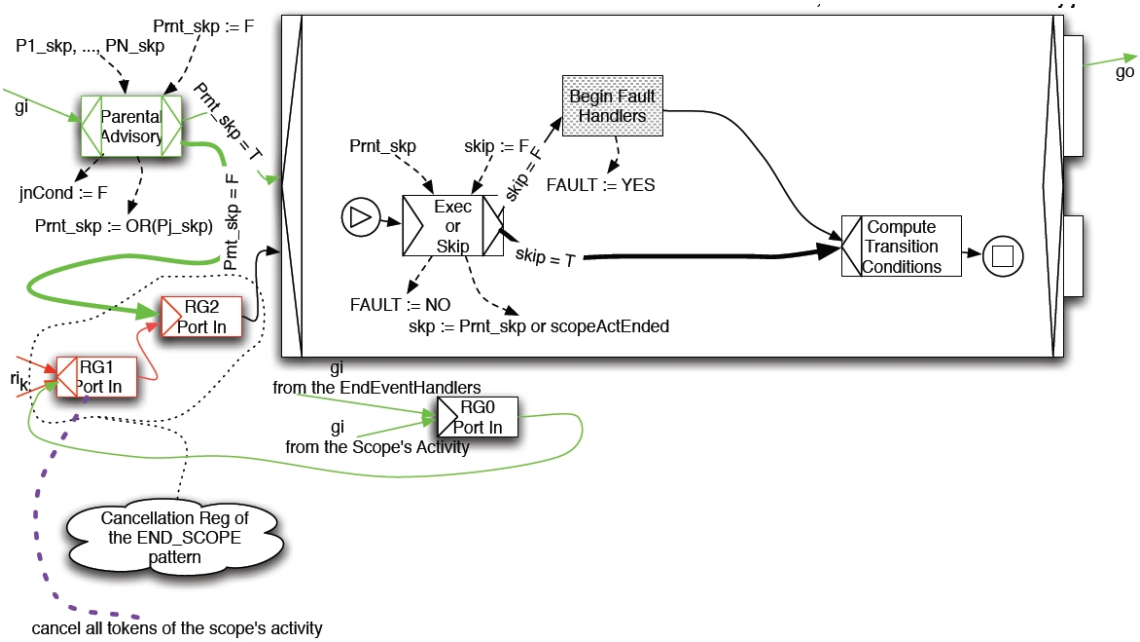


Figura 4.31: il pattern Begin Fault Handlers

La figura mostra il **Begin Fault Handlers**. Come abbiamo detto se la **`scope`** deve essere saltata la **Begin Scope** manderà un token al **Parental Advisory** del gestore, il quale eseguirà il **Main Task** ma, in seguito al salto del pattern dell'*attività* interna alla **`scope`** che viene eseguita in parallelo, arriverà un token da questa e si fermerà nel gruppo di task **RG1/RG2**. Questo token, come abbiamo detto, verrà cancellato dall'**End Scope**.

Da notare che la **Exec or Skip** della **Begin Fault Handlers** tiene in considerazione anche la variabile **scopeActEnded** in quanto se l'*attività* interna della **`scope`** ha avuto buon esito, bisogna applicare la *dead-path elimination* all'interno del **`faultHandlers`**.

Il seguente schema mostra come deve essere modificato il pattern di una *attività* interna ad una **`catch`**/**`catchAll`**:

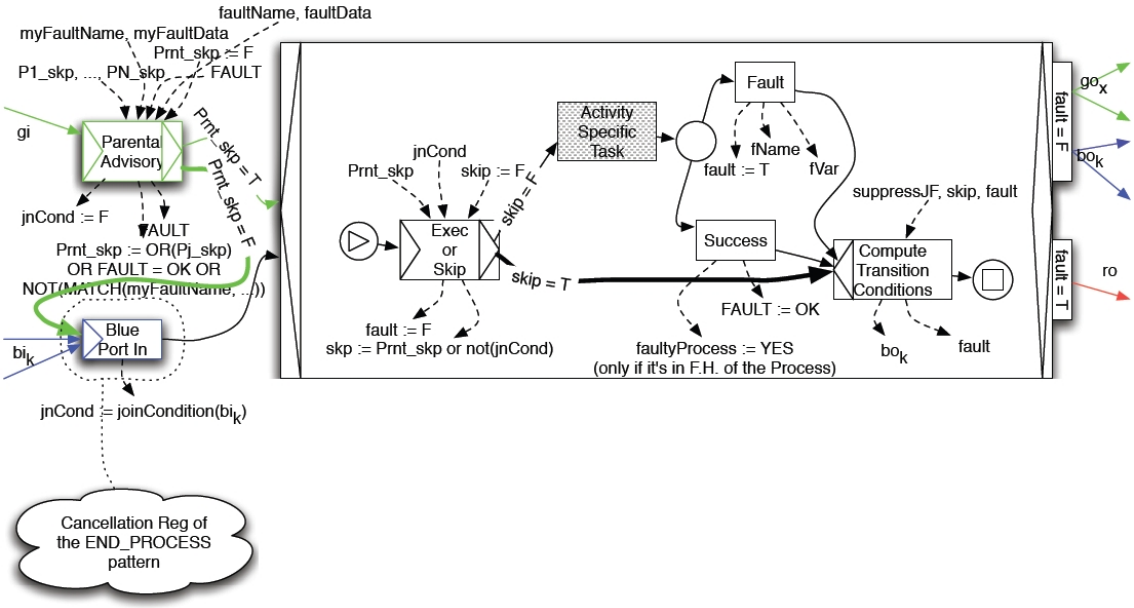


Figura 4.32: il pattern generico modificato dalla Catch

La **Parental Advisory** riceve in input quale tipo di fallimenti deve gestire (le variabili **`faultName`** e **`faultVariable`** della **`catch`** del documento BPEL4WS, che qui prendono il nome di **myFaultName** e **myFaultData**) e le variabili col nome e la variabile del fallimento generato (**faultName** e **faultData**). Per calcolare il valore della **skip** interna al pattern si valuta se questa *attività* va saltata, se il fallimento è già stato gestito (**FAULT=OK**) o non si è proprio verificato (**FAULT=NO**) e se il nome e la variabile del tipo di fallimento coincidono (da notare che uno tra il nome e la variabile potrebbe non essere definito, oppure entrambi nel caso di una **`catchAll`**, nel qual caso basta che coincidano i campi definiti).

Da notare inoltre che le variabili **FAULT** e **fault** sono diverse: la prima indica lo stato di gestione di un fallimento a livello di **`scope`**, la seconda se c'è un fallimento interno al pattern. Questo si può verificare anche durante l'esecuzione dell'*attività* interna ad una **`catch`**/**`catchAll`**, nel qual caso la gestione viene passata al **`faultHandlers`** dello **`scope`** padre o del processo intero. Se invece la gestione e l'esecuzione del pattern dell'*attività* interna dà buon esito, il task **Success** lo segnalerà tramite la variabile **FAULT** (prenderà il valore **'OK'**).

Da notare che se il **`catch`**/**`catchAll`** che stiamo trattando è quello dell'intero

processo, come vedremo nel prossimo paragrafo, sarà presente una variabile **faultyProcess** che indica se il processo non ha potuto terminare l'esecuzione a causa di un fallimento. In questo caso quando tale fallimento viene gestito l'*attività* che lo gestisce segnala che il processo non ha potuto terminare l'esecuzione mettendo a **YES** la variabile **faultyProcess**.

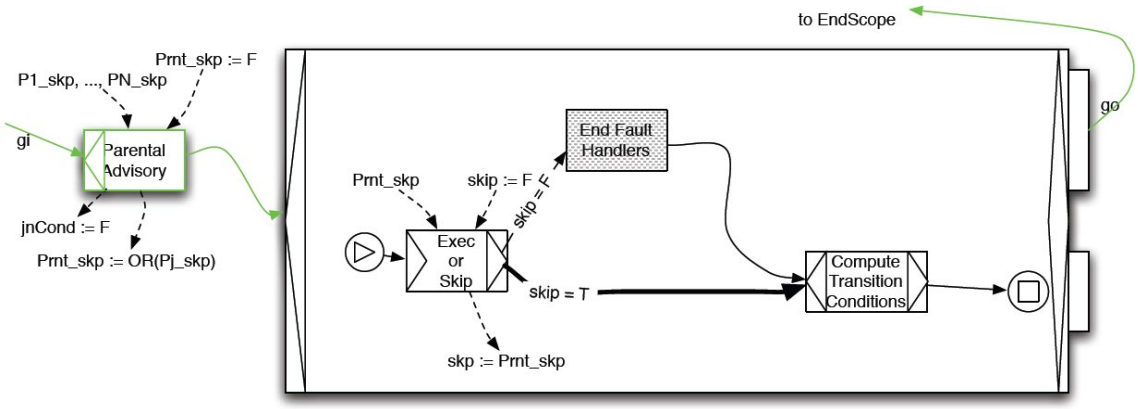


Figura 4.33: il pattern End Fault Handlers

End Fault Handlers è molto semplice e simile a molti altri pattern finali delle *attività* composite. Ricordiamo che il **green input** è ricevuto nei seguenti due casi:

- è stato ricevuto un fallimento che può essere stato gestito o no, nel qual caso si manda un token al pattern della **End Scope** che deciderà cosa fare (se mandare un **red token** al **`faultHandlers`** superiore oppure se continuare con l'esecuzione se il fallimento è stato gestito);
- il **`faultHandlers`** è stato saltato perché tutta l'*attività* è stata saltata o perché la **`scope`** è stata completata con successo, nel qual caso verrà comunque mandato un token all'**End Scope** che proseguirà l'esecuzione del processo.

Lo schema seguente mostra il pattern dell'**`eventHandlers`** ad alto livello:

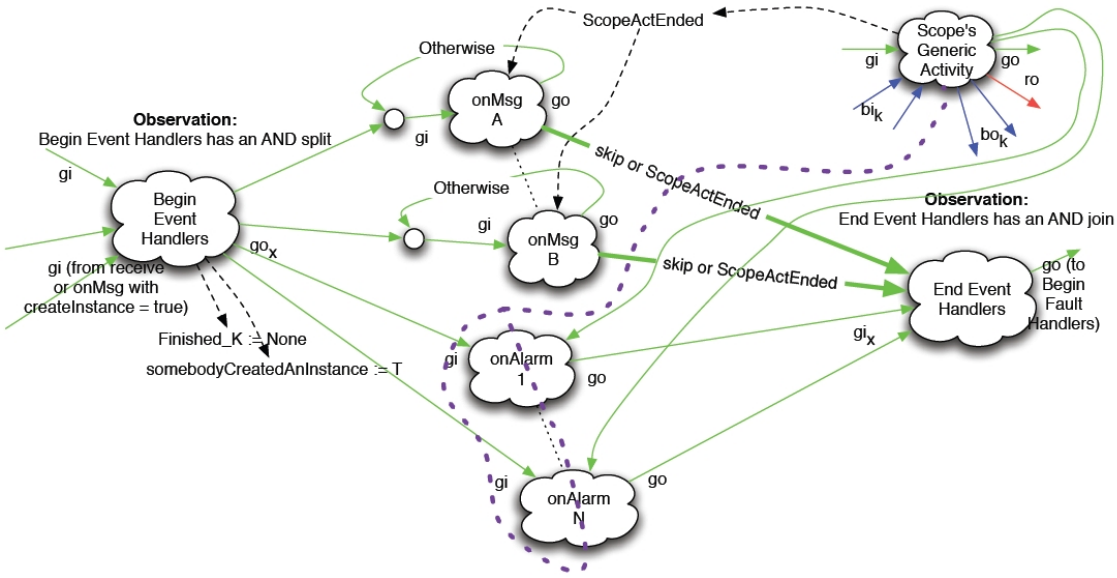


Figura 4.34: l'Event Handlers ad alto livello

Il primo pattern che analizzeremo sarà chiaramente il **Begin Event Handlers**. La versione mostrata rappresenta il pattern usato per il processo generale, e non quello della **`scope`**. Questo perché quello della **`scope`** risulta più semplice in quanto non ha bisogno di attendere che l'istanza di processo venga creata per andare in esecuzione.

Il pattern è il seguente:

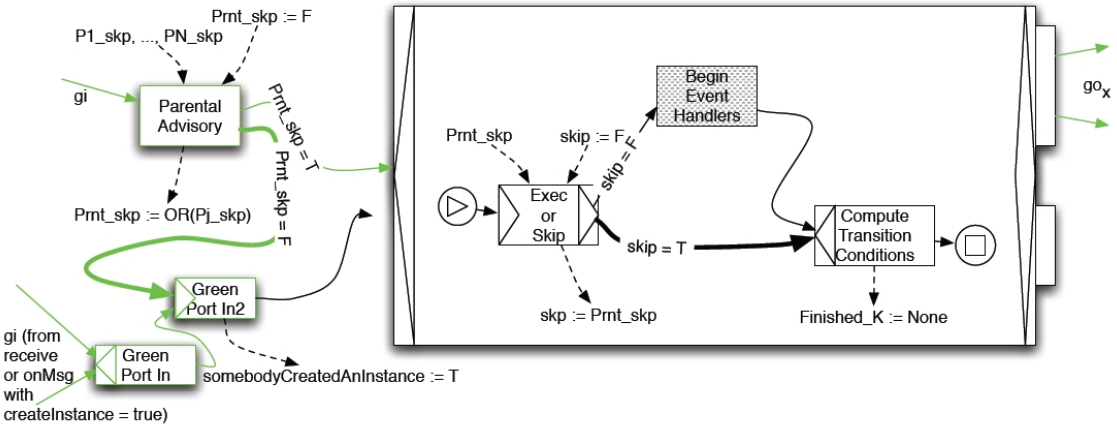


Figura 4.35: il pattern Begin Event Handlers

Solo un pattern di un **`onMessage`** o di una **`receive`** può mandare il proprio token alla

Green Port In poiché subito dopo viene modificato il valore della variabile **somebodyCreatedAnInstance** che blocca l'invio di ulteriori token. Questo metodo garantisce che finchè qualche *attività* non avrà creato una istanza di processo non si potrà eseguire il pattern **Begin Event Handlers**.

Come notato precedentemente non è però necessario bloccare il token nel caso di un **`eventHandlers`** interno ad uno **`scope`**. Il pattern in questo caso sarà lo stesso ad eccezione dell'esistenza dei due **Green Port In**.

L'**`eventHandlers`** deve eseguire l'*attività* specificata da una delle sue **`onMessage`** ogni volta che, durante il periodo in cui il processo è in esecuzione, il messaggio di cui è in attesa viene ricevuto. Questo significa che se viene ricevuto più volte un messaggio di un certo tipo è possibile che venga eseguita più volte una specifica **`onMessage`**. Per questo, come è mostrato in figura, le **onMsg**, pattern che simula una specifica **`onMessage`**, rimandano il loro token all'inizio del pattern dopo ogni esecuzione. Solo quando lo **`scope`** ha terminato la sua esecuzione e lo ha segnalato tramite la variabile **scopeActEnded** allora tutte le **onMsg** terminano la loro esecuzione mandando il loro **green token** all'**End Event Handlers**.

Il blocco della **onMsg** può essere quindi visto, ad un livello leggermente più basso, come un pattern composto come nella seguente figura:

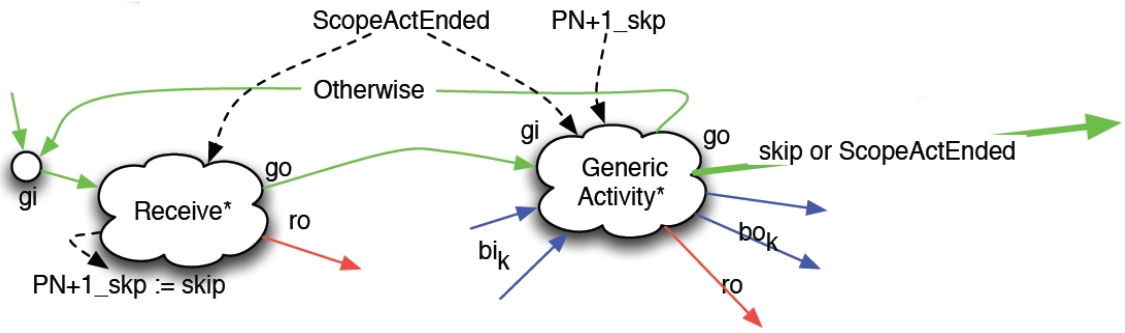


Figura 4.36: la onMessage ad alto livello

Come si può vedere la **onMsg** in realtà è divisa in due pattern: il primo, molto simile ad una **`receive`**, aspetta la ricezione del messaggio; il secondo implementa il corpo

della **`onMessage`** e risulta essere una *attività* generica modificata per comportarsi nella maniera che abbiamo descritto in precedenza.

La **Receive** è del tutto identica al pattern di una **`receive`** ad esclusione delle seguenti modifiche: il **Parental Advisory** riceve in input lo **ScopeActEnded** e lo utilizza per calcolare se deve o no saltare il pattern. Il task **Compute Transition Condition** viene modificato in modo che invii l'informazione di salto alla **Generic Activity,** che costituisce il corpo della **`onMessage`**, e fa questo tramite la variabile **PN+1_skp**. Questa *attività* a sua volta, come mostrato in figura, potrà essere eseguita e mandare il token all'inizio del pattern oppure essere saltata e mandare il token alla **End Event Handlers**. In base a quanto detto il pattern dell'*attività* generica non viene modificato se non per rimandare indietro il token, e il suo **Parental Advisory** prende in input il **PN+1_skp** calcolato precedentemente.

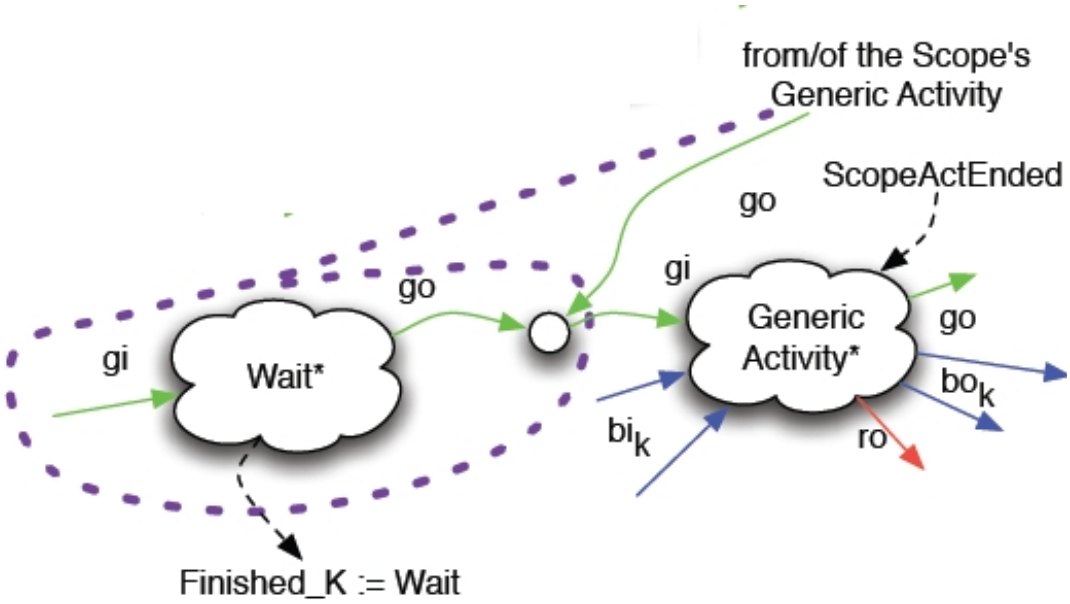


Figura 4.37: la onAlarm ad alto livello

Quello proposto sopra è lo schema di traduzione di una **`onAlarm`**. Il funzionamento è semplice: i task **`wait`** hanno associata una variabile **Finished_K** dove k indica che stiamo parlando del k-esimo **`wait`**. Quando la **`wait`** è completata cambia il valore di questa variabile da '**None**' all'id della **`wait`** e passa il token alla **Generic Activity**

(passando per la **`condition`**, che verrà utilizzata per saltare la **`onAlarm`** e non bloccare l'**`eventHandlers`** non appena la **`scope`** completata la sua esecuzione).

La **Generic Activity** calcola se deve essere saltata basandosi anche sul valore di questa variabile. Se la **`wait`** non ha avuto modo di cambiare il valore della variabile (non è quindi stata eseguita), quando l'*attività* interna della **`scope`** manderà il token, la **Generic Activity** verrà saltata perché non troverà la variabile col giusto valore, e si potrà eseguire la *dead-path elimination*.

La **Generic Activity** della **`scope`** non subisce molte modifiche: in caso di successo (ovvero se viene eseguito il task **Success** se presente o la **Activity Specific Task** nel caso il pattern non preveda la generazione implicita di fallimenti) deve soltanto essere attribuito alla variabile **scopeActEnded** il valore '**true**'. Questo pattern poi deve mandare un token alla **Begin Fault Handlers** (ricordiamo che questo token, nel caso in cui l'*attività* sia saltata, si ritroverà bloccato nella coppia di task **RG1** / **RG2 Port In**, ma verrà eliminato dalla **End Scope**, mentre se l'*attività* non deve essere saltata e non genera errori il token farà saltare comunque il **`faultHandlers`**) e alle **`wait`** dell'**`eventHandlers`**.

Per finire è necessario considerare il pattern della **`compensationHandler`**. Abbiamo accennato al suo funzionamento parlando della **`compensate`** nel paragrafo *4.1.3.8*. Nella nostra traduzione il **`compensationHandler`** viene quasi visto come un elemento interno delle varie **`compensate`**.

Questo è il pattern iniziale:

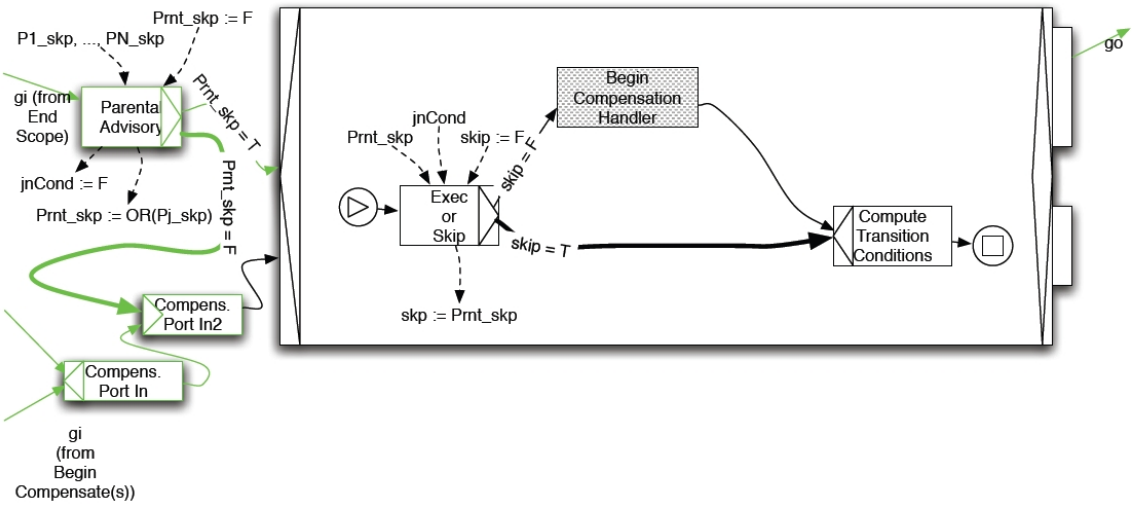


Figura 4.38: il pattern Begin Compensation Handler

Il token che arriva al **Parental Advisory** ha lo scopo, come abbiamo detto, di *attivare* il **`compensationHandler`** e viene ricevuto solo nel caso in cui la **`scope`** si sia conclusa con successo (quindi senza salti né errori). In caso contrario il **`compensationHandler`** non sarà installato e non potrà essere invocato, e questo comportamento sarà ottenuto bloccando eventuali token nel **Compensation Port In2**.

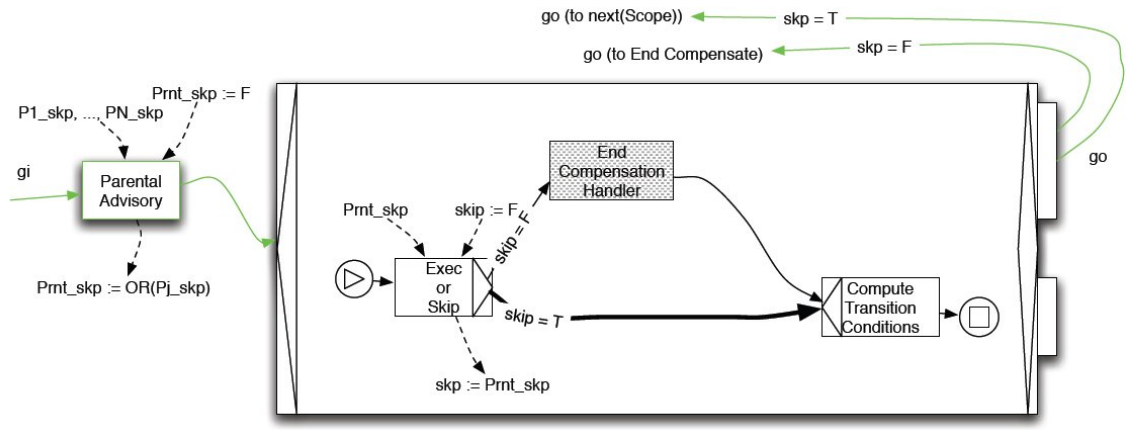


Figura 4.39: il pattern End Compensation Handler

L'*attività* della **`compensationHandler`** viene tradotta mediante il suo pattern solito, senza modifiche. Anche la **End Compensation Handler** usa un pattern identico a quello generalmente usato per rappresentare la parte finale delle *attività* composite. Questo poi manda un token alla **End Compensate** se non c'è stato salto di *attività*,

altrimenti manda un token all'*attività* successiva alla **`scope`**.

4.1.4 Il processo BPEL4WS

Ora che sono state introdotte tutte le *attività* con i relativi pattern di traduzione possiamo dare una visione completa di un processo BPEL4WS. Ad alto livello ci sono molte analogie con una **`scope`**; qui di seguito viene presentato lo schema:

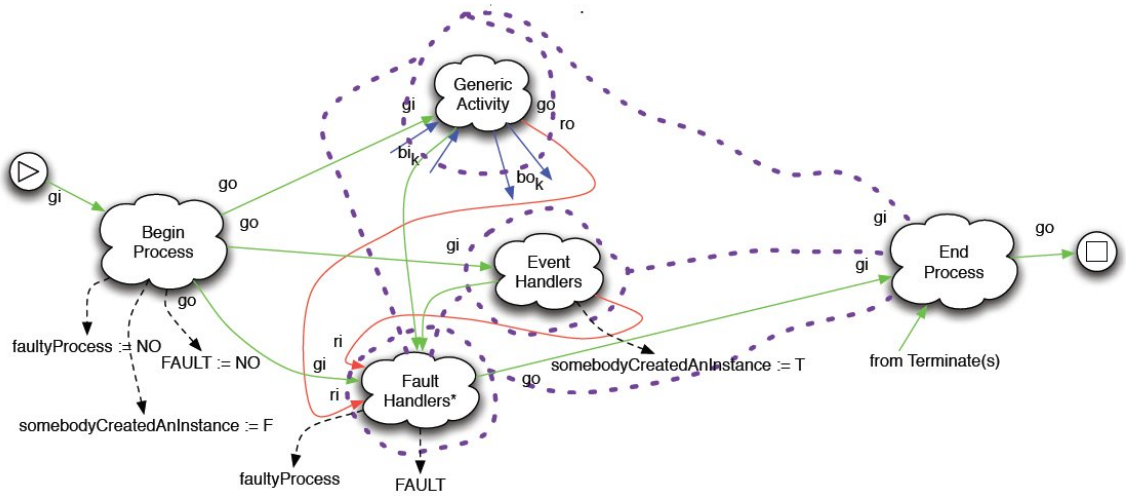


Figura 4.40: il processo BPEL4WS ad alto livello

Le differenze con i pattern della **`scope`** sono:

- il pattern **End Process** deve cancellare tutti i token del processo, mentre il **Begin Fault Handlers** deve cancellare tutti i token della **Generic Activity** del processo solo quando riceve un **red input**;
- il processo non ha un **`compensationHandler`** in quanto per la simulazione non ha senso compensare l'intero processo;

La variabile globale booleana **somebodyCreatedAnInstance** assume il valore '**true**' quando viene eseguita una *attività* col campo **`createInstance`** con il valore **`yes`** (una **`receive`** o una **`pick`**), nel qual caso all'**`eventHandlers`** del processo (e solo a questo) viene mandato un token di 'attivazione'.

La variabile globale booleana **faultyProcess** assume il valore '**true**' nel caso sia generato un fallimento che deve essere gestito nel `faultHandlers` del processo (quindi se la variabile **FAULT** a livello di processo assume il valore **YES** o **OK**). Questa variabile indica se l'esecuzione del processo è andata o no a buon fine.

La variabile **FAULT** indica se l'esecuzione sta procedendo senza fallimenti (**FAULT=NO**), se c'è da gestire un fallimento (**FAULT=YES**) oppure se c'è stato un fallimento che è stato gestito (**FAULT=OK**).

Se il documento BPEL4WS non definisce un `faultHandlers` per il processo, o lo definisce senza elemento `catchAll`, è compito del traduttore generarne uno che abbia un elemento `catchAll` la cui *attività* interna sia una `empty`. Questo per permettere al `faultHandlers` di gestire qualsiasi fallimento generato all'interno del processo.

Qui di seguito analizziamo brevemente il pattern **BeginProcess**:

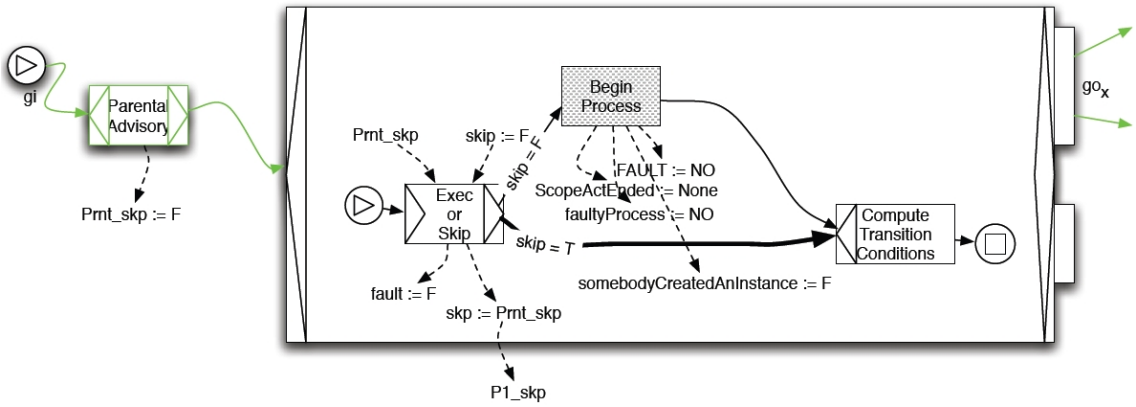


Figura 4.41: il pattern Begin Process

Tale pattern risulta molto semplice: non ha livelli di annidamento superiori, quindi non può essere saltato (abbiamo mantenuto il flusso che salta l'esecuzione della **Activity Specific Task** solo per analogia con gli altri pattern), il suo valore di **skip** risulta ovviamente '**false**' e viene trasmesso come **Prnt_skp** di tutto il processo.

Per il resto svolge il compito di inizializzare le variabili globali (**faultyProcess** e **somebodyCreatedAnInstance**), e anche quelle locali (**FAULT** e **ScopeActEnded**) utilizzate come se fosse una normale `scope`.

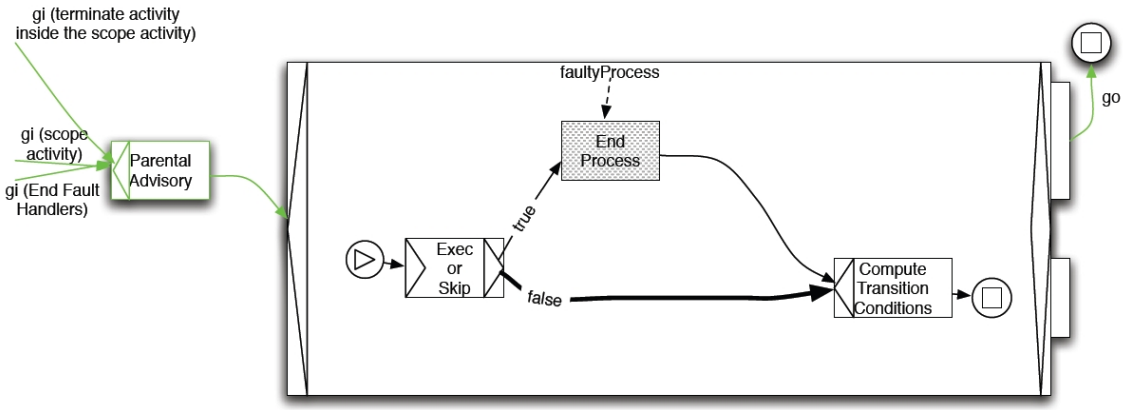


Figura 4.42: il pattern End Process

End Process attende di ricevere il token di terminazione mandato da una eventuale *attività* **`terminate`** oppure il token proveniente dal pattern della **`faultHandler`**.

Non c'è bisogno di controllare se un eventuale fallimento è stato gestito oppure no perché il **`catchAll`** deve esistere, quindi deve aver gestito ogni eventuale fallimento rimasto.

Per finire il compito della **End Process** è quello di cancellare tutti i token del processo, come spiegato precedentemente.

4.2 Realizzazione

Avere una traduzione tra i due linguaggi basata su pattern, ovvero poter stabilire che le varie attività BPEL verranno tradotte in uno o più task YAWL ben definiti, ci permette di usare un metodo abbastanza semplice per sviluppare il traduttore BPEL2YAWL. Tale metodo consiste nell'avere due gerarchie di classi, una per tipo di documento, la cui struttura sia simile alla gerarchia dell'albero XML degli elementi del documento corrispondente (questo ci permette, in fase di traduzione, di ricostruire facilmente il documento XML finale dell'uno e dell'altro tipo e ciò è utile, nel caso di BPEL4WS per il caricamento e nel caso di YAWL per il salvataggio), e nell'avere per le classi che gestiscono i documento BPEL4WS un metodo che ne faccia ottenere una rappresentazione YAWL e quindi, per le singole *attività* BPEL4WS, che restituisca il

pattern di traduzione corrispondente.

Il progetto si divide in due pacchetti, ognuno dei quali gestisce uno dei due tipi di documenti. I due pacchetti sono stati denominati BPELDoc (quello che gestisce i documenti BPEL4WS) e YAWLDoc (quello che gestisce i documenti YAWL). BPELDoc ha, tra le altre funzioni, quella di trasformazione che, andando a utilizzare le classi dell'altro pacchetto, permette di generare un documento trasformato (quindi un workflow YAWL).

Si è prestata una certa attenzione ai casi eccezionali: quindi laddove la specifica di BPEL4WS richiede la presenza di determinati attributi o elementi, la mancanza di tali elementi a livello di oggetti Java viene sottolineata con il lancio di una eccezione.

Il pacchetto BPELDoc, oltre a gestire i dati del documento BPEL4WS, permette, attraverso le sue funzioni, di ottenere un oggetto che modelli il documento YAWL, che rispetti la traduzione descritta nel paragrafo *4.1* e che possa, quindi, restituire il documento XML tramite le proprie chiamate di funzione.

Nei seguenti paragrafi analizzeremo i due pacchetti ad alto livello. Per quanto riguarda BPEL4WS (e quindi il pacchetto BPELDoc) dopo aver visto quale è la gerarchia delle classi e dopo aver discusso in che modo le classi principali lavorino per ottenere i requisiti richiesti si analizzerà brevemente la struttura della classe che modella il pattern generico e si discuterà come, grazie all'ereditarietà, sia possibile ottenere gli oggetti che rappresentano gli altri pattern.

4.2.1 Il package BPELDoc

Il package *BPELDoc*, come abbiamo già detto, gestisce un documento BPEL4WS permettendone il caricamento, il salvataggio e ovviamente la trasformazione in un documento YAWL. La classe principale, chiamata anche essa **BPELDoc**, fornisce queste tre funzionalità attraverso altrettanti metodi pubblici. La classe mantiene il riferimento ad un oggetto di tipo **BPELProcess**, che modella un processo BPEL4WS.

Come discusso nel paragrafo *2.1.5*, un processo BPEL4WS è formato da diversi elementi, che nel caso del package sono modellati da altrettante classi. Qui di seguito viene fornito un semplice diagramma UML che rappresenta la classe **BPELProcess** e

come questa è composta da oggetti di altre classi.

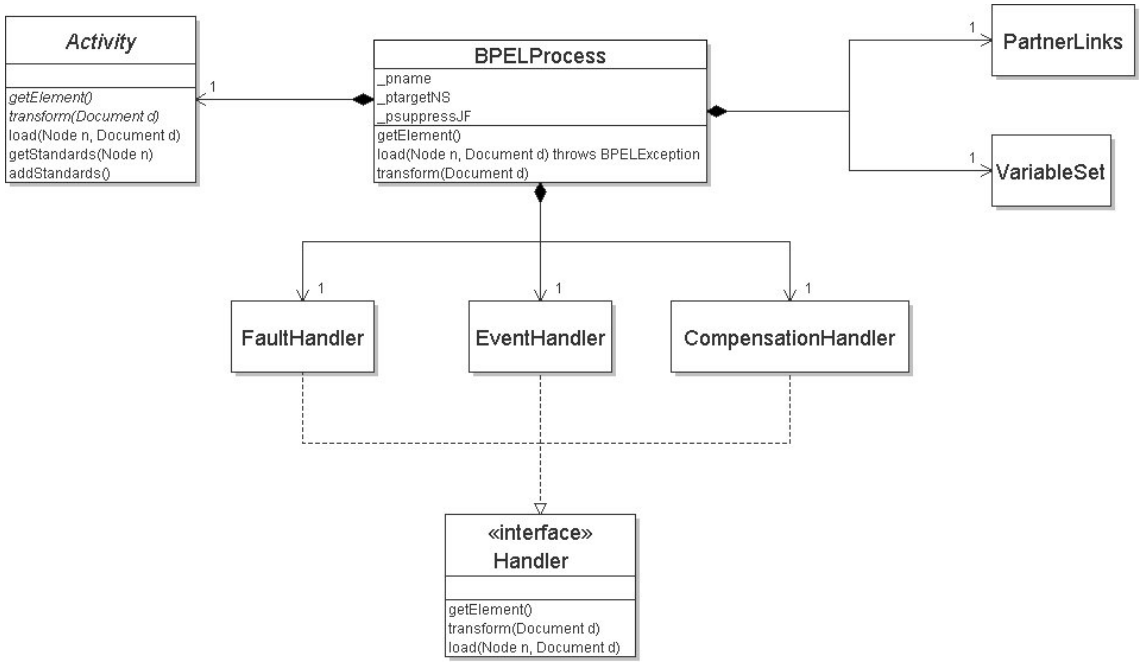


Figura 4.43: diagramma delle classi UML con classi che compongono la BPELProcess

È facile da questo schema intuire il significato delle classi da cui è composta la **BPELProcess**. Tale classe mantiene un riferimento ad una classe astratta, **Activity**, da cui ereditano tutte le classi che modellano le varie attività. Infine, oltre ai metodi elencati nel diagramma, ve ne sono altri per gestire i dati interni del processo.

PartnerLinks e **VariableSet** sono classi molto semplici, in pratica sono formate da una tabella hash che contiene come chiave nel primo caso il nome del partnerLink e nel secondo il nome della variabile, e come oggetto una istanza della classe **PartnerLink** nel primo caso e una della classe **BVariable** nel secondo. Le quattro classi appena citate (**PartnerLinks, PartnerLink, VariableSet** e **Variable**) hanno i metodi (statici) per generare un oggetto del loro tipo a partire da un elemento XML che li rappresenta in base alla sintassi BPEL4WS (metodo *load*) e per generare un elemento XML che li rappresenti (metodo *getElement*).

In linea generale tutte le classi che modellano una parte del documento BPEL4WS hanno questi due metodi: questo permette di svolgere queste due funzioni in maniera molto semplice, in quanto ogni oggetto si occupa di ottenere dal nodo XML che lo

rappresenta solo le poche informazioni che servono a se stesso e di lasciare l'interpretazione dei dati restanti ai propri figli, così per esempio la classe **BPELProcess**, in base a quanto detto nel paragrafo *2.1.5*, riceverà un elemento XML di tipo **`process`** e da questo estrarrà solo i dati principali (**`name`, `targetNamespace`, `queryLanguage`, `expressionLanguage`, `suppressJoinFailure`, `enableInstanceCompensation`, `abstractProcess`**), lasciando ai metodi statici delle classi che compongono i propri oggetti interni il compito di ottenere tutti gli altri dati.

Ovviamente, seguendo questa ottica, esiste tutta una gerarchia di classi che vanno a costituire le piccole parti che compongono un documento BPEL4WS, così la classe **FaultHandler** contiene un vettore di oggetti di tipo **CatchElement** che possono essere delle semplici **Catch** oppure **CatchAll,** come mostrato nel seguente diagramma UML.

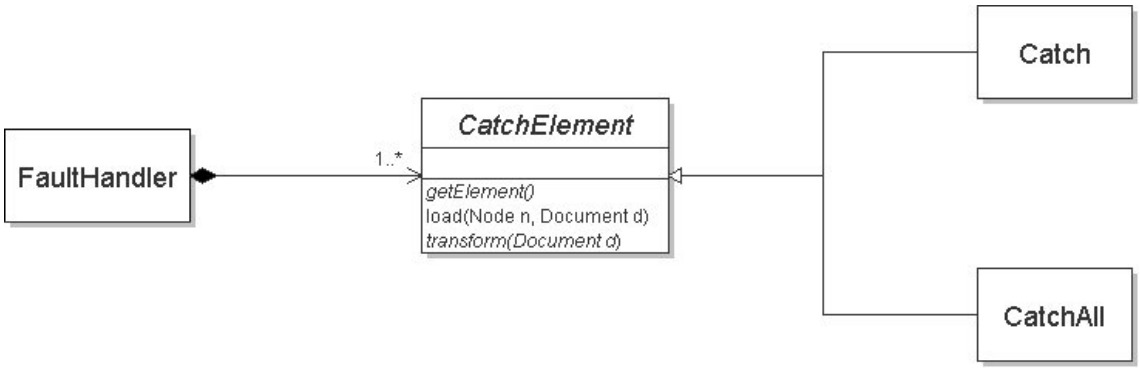


Figura 4.44: diagramma delle classi UML focalizzato sulla FaultHandler

CatchElement è una classe astratta che mantiene astratti tutti i metodi ad eccezione di uno, *load*, che quando viene invocato determina se l'elemento XML si riferisce ad una **Catch** o ad una **CatchAll** e ne invoca a sua volta il metodo *load*. Allo stesso modo ci si comporterà nella classe astratta **Activity** per determinare di quale classe bisogna invocare il metodo *load* tra tutte quelle che modellano le varie *attività*.

Ovviamente le altre classi di cui abbiamo parlato in precedenza non differiscono di molto, per quanto riguarda la loro struttura, da quanto visto sopra per il **FaultHandler**; ad esempio la classe **EventHandler** invece che dei **CatchElement** avrà un vettore di oggetti della classe **OnMsg** (che modella un elemento **`onMessage`**) e uno di **OnAlarm** (che modella l'omonimo elemento).

Altra parte del pacchetto importante da considerare è la classe astratta **Activity**. Abbiamo già parlato di come il suo metodo *load* serva a decidere a quale tipo di attività un elemento XML di un documento BPEL si riferisce, per poi invocarne il corrispondente metodo *load*. Tale classe astratta mantiene anche le variabili utili a salvare attributi e elementi standard (quelli di cui ho parlato nel paragrafo *2.6.1*), e ha due metodi (*addStandards* e *getStandards*) che servono a caricare o a restituire proprio questi elementi. Quindi nelle *load* di una attività specifica generalmente, subito dopo la creazione di un oggetto di quella classe, viene invocato il metodo *getStandards* che semplicemente legge i dati standard di una attività BPEL4WS (gli elementi dei *link* e gli attributi **`name`**, **`joinCondition`** e **`suppressJoinFailure`**) dal nodo XML e li salva nel corrispondente oggetto.

Le *attività* composte di BPEL4WS al loro interno contengono riferimenti ad altri oggetti **Activity**.

Per quanto riguarda la struttura generica delle classi e le loro dipendenze non c'è molto di più da dire: lo schema di funzionamento ad alto livello è semplice.

Ora resta solo da analizzare come è stata realizzata la classe che modella il pattern generico della traduzione di BPEL4WS, di cui abbiamo parlato nel paragrafo *4.1.1*.

4.2.1.1 La classe GenericActivity

La classe **Activity** modella ad alto livello una *attività* generica vista a partire dalle informazioni di BPEL4WS. Questa *attività* generica, come abbiamo detto, deve avere la possibilità di tradursi nei pattern YAWL, e quindi di poterne creare istanze.

Per questo motivo nell'**Activity** ci sono riferimenti a due oggetti della classe **GenericActivity,** la quale modella un pattern generico YAWL. Due riferimenti perché, come abbiamo visto, tutte le *attività* sono costituite da uno o due pattern. Bisogna notare che anche una *attività* come la **`scope`** che, come si può vedere nel paragrafo *4.1.4.6*, è composta da più pattern può però essere rappresentata solo dalla sua **Begin** e dalla sua **End**.

Infatti gli elementi "interni" ad un pattern, come possono essere gli **Handler** per una **`scope`**, saranno rappresentati da un loro pattern YAWL, e l'oggetto padre, come la **`scope`**, avrà un riferimento all'oggetto BPEL4WS che li modella.

Quindi abbiamo detto che ogni *attività* contiene due riferimenti a istanze della classe **GenericActivity**. Nel caso in cui l'*attività* sia tradotta con un unico pattern, le due variabili saranno inizializzate con lo stesso elemento. La classe **GenericActivity** a sua volta contiene i riferimenti a tutti gli oggetti che compongono il pattern generico, e tutte le variabili utili, ovvero:

- I task **_pa, _bpi, _main, _EoS, _Activity, _S, _F, _CTC** che corrispondono ai task **Parental Advisory, Blue Port In, Main Task, Exec or Skip, Activity Specific Task, Success, Fault** e **Compute Transition Condition**.
- La condizione **_toFault** che permette di scegliere l'esito dell'esecuzione di quelle attività che possono generare fault.
- La decomposizione del **Main Task**, che permette di creare la sottorete di cui abbiamo discusso quando abbiamo parlato del pattern generico.
- Le variabili per la **`suppressJoinFailure`**, la **`joinCondition`** e per la **Prnt_skp**.
- Altre variabili utili per la traduzione, tra cui un importante **_oid**, l'*object identifier* che permette di specificare in maniera univoca variabili e nomi di task.

Il metodo principale della classe è l'**initializeConversion** che ha come argomenti la stringa che verrà usata come **Prnt_skp** e quella della **`joinCondition`**. Il metodo, l'unico a non dover essere mai modificato nelle classi che ereditano da questa, dopo aver convertito la stringa della **`joinCondition`** in una forma più adatta a YAWL (modificando le chiamate alla funzione XPath **bpws:getLinkStatus** in modo tale da prendere i valori dei link direttamente come variabili della rete), chiama tutti i metodi della classe che descrivono il pattern stesso.

Di questi metodi ce ne sono due per ogni task, ad eccezione dei task del "*deferred choice*" (paragrafo *3.2* pattern 16) per il successo dell'*attività* (quindi sono esclusi i task **_S** e **_F**). Il primo di questi metodi (che ha come nome **set** seguito dal nome del task e da **Form**, come per esempio **setBluePortInForm**) si occupa di stabilire a quali altri task

del pattern il task in esame è collegato mediante un flusso, mentre il secondo (che ha come nome **set** seguito dal nome del task e da **Behaviour**, come ad esempio **setBluePortInBehaviour**) si occupa di definire le variabili del task in esame e i suoi *mapping* (si veda il paragrafo *3.4*).

In questo modo per le classi che ereditano da questa è semplice modificare il pattern generico: basta ridefinire alcuni di questi metodi nella maniera adeguata.

Oltre a quei metodi ce ne sono altri importanti: **resetSJF** che permette di definire la `suppressJoinFailure` oppure indicare da quale altra *attività* dovrà ereditare tale valore (passando come argomento il suo **_oid** che, come spiegato precedentemente, lo identifica univocamente; infatti esisterà nel workflow una variabile chiamata "**SuppressJF_**" seguita dal valore dell'**_oid**), **setActivitySpecificTask** che permette di inizializzare il task **_Activity** (questo metodo viene ridefinito in tutte le *attività* per dare un nome a tale task, mentre non sempre vengono ridefiniti i metodi **setActivityForm** e **setActivityBehaviour**), **setFault** che definisce i collegamenti e il comportamento dei due task **_S**, **_F** e della condizione **_toFault**, oltre a collegare il **Main Task** con il gestore degli errori, **setCancellation** che aggiunge i task al *cancellation set* (vedere paragrafo *3.3*) del gestore degli errori, i due metodi **linkTo** e **linkFrom** che modificano il task nel caso sia la sorgente (il primo) o la destinazione (il secondo, che in realtà si occupa solo di definirne la `joinCondition` standard qualora non ne fosse stata definita una specifica) di *link* e la **addFlow** che collega i pattern tra loro.

Con il pattern generico modellato in questo modo dalla classe **GenericActivity** risulta semplice ridefinire tutti i pattern delle altre *attività*. Avendo già discusso le differenze tra il pattern generico e gli altri nel paragrafo *4.1* le classi che modellano gli altri pattern non verranno esaminate.

A scopo esemplificativo verrà discussa solo la classe **EmptyActivity**, che modella il pattern della `empty`, perché è un pattern che differisce poco da quello iniziale, ma che compare spesso come parte interna di altri pattern. Questa classe, oltre a definire il costruttore invocando quello della **GenericActivity** ma definendo a null il *deferred choice* (task **_S**, **_F** e condizione **_toFault**), ridefinisce solo tre metodi: la

setActivitySpecificTask, che, come abbiamo detto, viene ridefinita in tutte le classi che ereditano per inizializzare il task **_Activity**, la **setActivityForm** che collega l'**_Activity** con la **_CTC** (ovvero la **Compute Transition Condition**), e per finire la **setFault** che, a differenza del pattern generico che collegava il **_toFault** coi task **_S** e **_F** e questi al **_CTC**, non esegue nulla a parte il collegamento del **Main Task** al gestore degli errori.

Gli altri pattern vengono modellati in maniera simile, ridefinendo solo i metodi necessari.

4.2.2 Il package YAWLDoc

Questa libreria è stata sviluppata in maniera del tutto analoga a quella precedente. Nonostante sia stata implementata una funzione di caricamento di documenti YAWL, funzione non necessaria al fine dello sviluppo del traduttore ma molto utile in fase di test, la libreria **YAWLDoc** ha una complessità di gran lunga inferiore a quella che gestisce i documenti BPEL4WS, sia perché il linguaggio (e i documenti XML esaminati nel paragrafo *3.6*) risulta decomponibile in un minor numero di 'elementi' (ognuno dei quali è stato modellato da una classe), sia per il minor numero di funzioni necessarie per il traduttore.

Infatti le classi, oltre ad avere le funzioni specifiche (task e condizioni avranno la possibilità di 'collegarsi' tra loro mediante flussi, una **Decomposition** avrà la possibilità di definire le proprie variabili e così via), devono avere una sola funzione (due, se contiamo il caricamento), ovvero quella che restituisce una parte di documento XML che le rappresenti.

Il seguente diagramma UML rappresenta la struttura di tutta la libreria:

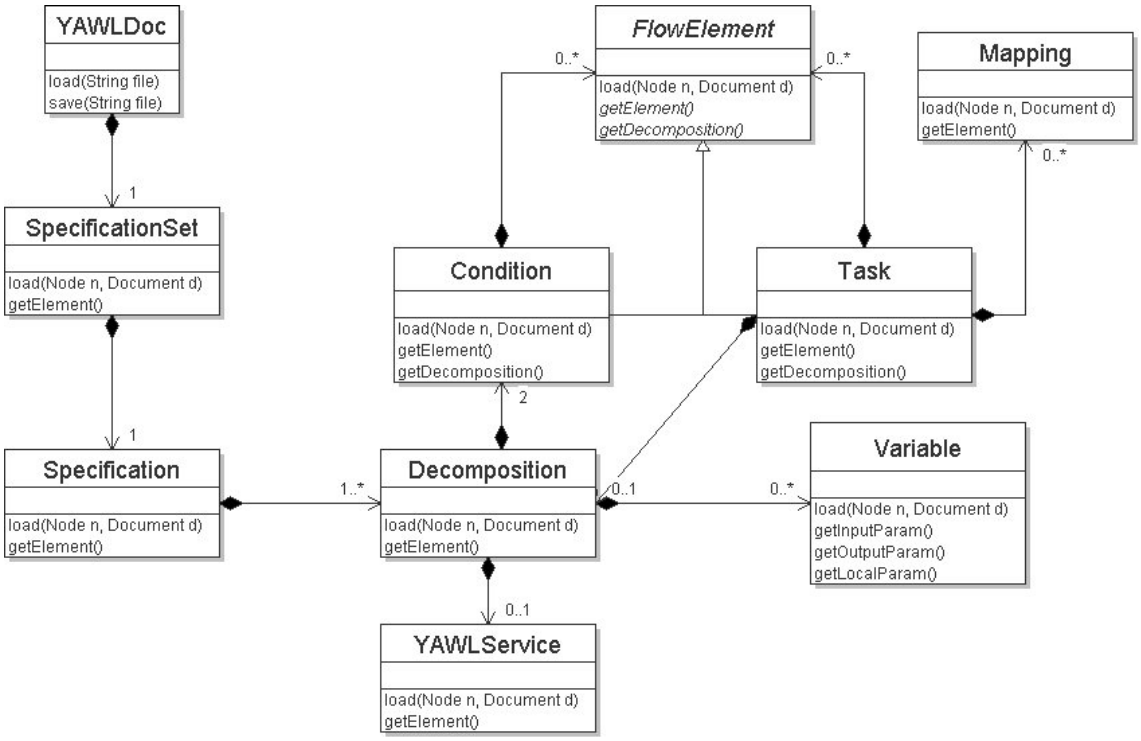


Figura 4.45: diagramma delle classi UML per il package YAWLDoc

La classe principale di tutta la libreria è **YAWLDoc**. Questa, oltre al costruttore, definisce solo due metodi pubblici: **load** che, presa la stringa col riferimento al nome del file, carica tale documento e ne effettua il parsing generando l'albero di oggetti che ne rappresenta il workflow, e **save** che, preso il nome del file su cui salvare i dati, genera il documento XML che lo rappresenta e che può essere caricato dall'engine di YAWL.

Gli oggetti di questa classe tengono un riferimento ad un oggetto di tipo **SpecificationSet**, la cui classe, a sua volta, mantiene un riferimento ad un solo oggetto di tipo **Specification**.

È facile notare che la struttura gerarchica delle classi rispecchia quella del documento XML che viene generato dall'editor quando si esporta un workflow per farlo caricare all'engine. In effetti, contando la semplicità della classe sarebbe stato possibile incorporare la classe **SpecificationSet** all'interno della YAWLDoc, aumentando però molto la complessità della sua funzione di salvataggio. Questo modo di procedere rende

invece tutta la libreria molto modulare.

La classe **Specification** è un contenitore di **Decomposition**. Ne viene identificata una come **root** (radice, vedere paragrafo *3.3*), e tutte le decomposizioni (compresa la **root**) sono salvate in una tabella hash, la cui chiave è l'identificativo unico della decomposizione (servirà poterle riferire quando si gestiranno le decomposizioni dei task) . All'interno di una **Specification** deve essere definita almeno una **Decomposition**, quella di **root**. Altre possono essere definite per specificare i dati di una sottorete o di un template per un **Task**.

La classe **Decomposition** è composta da due **Condition**, quella di input e quella di output, può contenere delle variabili e un servizio. La **Decomposition** modella l'omonimo elemento del documento XML che descrive un workflow YAWL, di cui abbiamo discusso nel paragrafo *3.6*.

Per quanto riguarda **FlowElement**, è una classe astratta che modella un generico elemento che fa parte del flusso di un workflow. Tutti gli oggetti delle classi che derivano da questa possono mandarsi token tra di loro. In pratica derivano da questa classe sia la classe **Task** che la classe **Condition**. Da notare che gli oggetti di entrambe queste classi tengono in un vettore (chiamato **_flowsInto**) i riferimenti agli altri oggetti a cui possono mandare il proprio token.

Ricordiamo che una proprietà degli elementi di flusso di un workflow YAWL è che ci deve essere sempre almeno un cammino che li comprenda e che parta dalla condizione di input per arrivare a quella di output (come discusso nel paragrafo *3.3*). Mantenere i riferimenti in questa maniera gerarchica ci permette di controllare facilmente che tale proprietà sia verificata, in quanto se un task non fa parte di un cammino che parte dalla condizione di input, o lui o uno dei task precedenti non faranno parte del vettore **_flowsInto** di nessun task, e quindi una volta creati ne verrà perso il riferimento e ci penserà il Garbage Collector di Java a rimuoverli. E per controllare che il flusso sia collegato alla condizione di output basterebbe che tutti gli oggetti controllassero se tra gli elementi restituiti dal metodo **getElement** ci fosse quello che

rappresenta la condizione di output. Tuttavia, per come viene gestita questa libreria all'interno del progetto, non c'è stato bisogno di implementare questa verifica.

Per quanto riguarda la classe **Task** è forse quella più complessa. Ha due vettori che tengono riferimenti ad oggetti **Mapping** che modellano i *mapping* (vedere paragrafi *3.4* e *3.6*) in ingresso e in uscita dal task. Ha il vettore **_flowsInto** e può anche avere il riferimento ad una **Decomposition** che ne determina il template dei dati.

Per quanto riguarda la creazione del file XML che abbia un formato che rispecchia quello descritto nel paragrafo *3.6*, e utile in fase di salvataggio (invocazione della funzione **save**), ogni oggetto costruisce il proprio elemento dell'albero XML creando un elemento XML che contenga i propri dati e appendendoci poi, ovviamente nella giusta posizione, gli elementi ottenuti invocando la **getElement** dei propri figli.

Questo modo di procedere necessita delle seguenti attenzioni:

- Più task possono riferire la stessa decomposizione e le decomposizioni nei documenti XML di YAWL sono ad un livello più alto dei task (da qui la necessità di ottenerle con un metodo separato).
- In caso di cicli all'interno del workflow bisogna evitare di chiamare in maniera ciclica le funzioni **getElement**, in più se due **FlowElement** hanno nel proprio flusso lo stesso oggetto bisogna che entrambi lo segnalino come destinazione del proprio flusso, ma la descrizione dell'elemento deve comparire solo una volta nel workflow;
- Le variabili che sono di input e output contemporaneamente devono comparire nella stessa decomposizione sia come variabili di input che come variabili di output. In più nel documento XML compaiono prima tutte le definizioni di variabile di input, poi quelle di output e infine le locali. Per questo la classe **Variable** ha tre metodi **get**.

Risolti questi problemi abbiamo che il nostro pacchetto YAWLDoc modella in maniera adeguata un workflow YAWL e restituisce documenti XML conformi ai documenti standard che vengono generati dall'editor.

4.2.3 Esecuzione di BPEL2YAWL

In questo paragrafo finale esamineremo tutti i passi compiuti durante la traduzione di un documento BPEL4WS, così da chiarificare ulteriormente il funzionamento del progetto. Si tengano sempre presenti i paragrafi *4.2.1* e *4.2.2* per avere i riferimenti alle librerie e alle sue classi.

Quando viene lanciato il programma su un documento BPEL4WS, il primo passo è la chiamata del metodo **load** sul nome del file XML che contiene la descrizione del processo. La **load** si costruisce un **DocumentBuilder** (che è un oggetto della libreria *javax.xml.parser*) che fa il parsing del documento XML, poi invoca il metodo **load** della classe **BPELProcess**, che, come abbiamo detto, modella i dati generali del processo.

Come abbiamo già spiegato la struttura del pacchetto BPELDoc rispecchia la struttura del documento XML, quindi se **BPELProcess** mantiene i dati dell'elemento **`process`** del documento BPEL4WS, delegherà alla classe **PartnerLinks** la gestione dell'elemento XML **`partnerLinks`** , alla classe **VariableSet** la gestione dell'elemento **`variables`** , e così via.

Queste classi richiameranno a loro volta altre classi per gestire altri sottoelementi dell'albero XML.Quando tutti gli oggetti a cui **BPELProcess** ha delegato il lavoro avranno completato, anche il metodo **load** di **BPELProcess** sarà completo, e a **BPELDoc** verrà restituito un oggetto **BPELProcess**. Questo completa il primo passo di traduzione.

Il secondo passo consiste nell'invocare il metodo dell'oggetto **BPELDoc** chiamato **transformToYawlDoc**. Questo crea un oggetto **YAWLDoc**, ci attribuisce un oggetto **SpecificationSet** e invoca la **transform** del proprio oggetto **BPELProcess**, che restituirà una **Specification** che rappresenta il workflow YAWL, e che quindi dovrà

essere collegata alla **SpecificationSet**. Quando sarà stato fatto quanto detto si avrà un documento YAWL perfettamente composto, e basterà invocarne il metodo **save** che provvederà a richiamare tutti i propri oggetti in maniera gerarchica per farsi restituire gli elementi XML e, costruito una rappresentazione dell'albero XML adeguata, tramite l'oggetto **TransformerFactory** (della classe *javax.xml.transform*) verrà salvato su un file.

Rimane da spiegare come funziona la trasformazione dell'oggetto **BPELProcess**. Questo dapprima si crea un oggetto **Specification** e un **Decomposition** (sarà la decomposizione principale che fornità la rete *radice* di tutto il workflow) che poi verranno collegati assieme, poi istanzia un pattern **Begin Process** e uno **End Process**, si crea un **Fault Handler** standard se non ne è stato definito uno e poi genera il pattern del **Fault Handler** e lo collega al **Begin Process** e all'**End Process**. Dopo genera il pattern anche dell'**Event Handler**, e collega anche questo.

Poi, dopo aver inserito nella **Decomposition** principale la traduzione di tutte le variabili trasforma l'*attività* principale (che è un oggetto **Activity**) in un pattern e ne collega la **Begin Process** con la **start** e la **end** con la **End Process**. Questi due elementi *potrebbero* essere lo stesso elemento se l'*attività* principale fosse una *attività* semplice. Tale fatto avrebbe poco senso, ma sarebbe comunque corretto, generando una sequenza di **Begin Process – Attività – End Process**. Questa precisazione è importante perchè la traduzione viene fatta in maniera gerarchica, cioè, se l'*attività* principale fosse ad esempio una `flow` a sua volta questa istanzierebbe un pattern iniziale **(Begin Flow)** e uno finale **(End Flow)**, poi trasformerebbe le proprie *attività* interne e le aggancerebbe a queste due usando lo schema **Begin Flow – Activity.start – Activity.end – End Flow**, generando così il pattern come descritto nel precedente paragrafo *4.1*.

Una volta creato il workflow rimane da gestire alcune particolarità che non potevano essere gestite durante una trattazione di questo tipo: i *link* (quando viene trovata una sorgente o una destinazione non si può dire se la controparte sia già stata generata, quindi si rimanda alla fine il compito di collegare le attività in questo modo) e collegare le `compensate` ai gestori (anche in questo caso potevano non esistere ancora). Per

questo motivo sono state create delle strutture dati apposite (tipicamente o vettori o tabelle hash) per mantenerne i dati. Completato anche questo passaggio rimane solo da collegare la condizione di input della **Decomposition** principale con la **Begin Process** e la **End Process** con la condizione di output.

4.3 Esempio

In questo ultimo paragrafo di questo capitolo illustreremo un esempio pratico. Non è semplice mostrare il risultato della traduzione YAWL attraverso il file generato dal traduttore, poiché questo risulta essere praticamente illeggibile e di dimensioni poco pratiche. Per questo motivo ne verrà presentata qui una esemplificazione grafica.

Intanto partiamo presentando il documento BPEL4WS che vogliamo tradurre. Questa è la sua versione testuale:

```
    <process name="ExampleBPEL4WS2YAWL"
targetNamespace="http://xmlns.oracle.com/ExampleBPEL4WS2YAWL"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:xp20="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.Xpath20"
xmlns:ldap="http://schemas.oracle.com/xpath/extension/ldap"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bpelx="http://schemas.oracle.com/bpel/extension"
xmlns:client="http://xmlns.oracle.com/ExampleBPEL4WS2YAWL"
xmlns:ora="http://schemas.oracle.com/xpath/extension"
xmlns:orcl="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.ExtFunc">
    <!-- ================================================================= -->
    <!-- PARTNERLINKS--><!-- List of services participating in this BPEL process -->
    <!-- ================================================================= -->
      <partnerLinks><!--
        The 'client' role represents the requester of this service. It is
        used for callback. The location and correlation information associated
        with the client role are automatically set using WS-Addressing.
        -->
        <partnerLink name="client" partnerLinkType="client:ExampleBPEL4WS2YAWL"
myRole="ExampleBPEL4WS2YAWLProvider"/>
      </partnerLinks>
    <!-- ================================================================= -->
    <!-- VARIABLES--><!-- List of messages and XML documents used within this BPEL process  -->
    <!-- ================================================================= -->
      <variables><!-- Reference to the message passed as input during initiation -->
        <variable name="inputVariable"
messageType="client:ExampleBPEL4WS2YAWLRequestMessage"/><!--
        Reference to the message that will be returned to the requester
        -->
        <variable name="temporaryVar"
messageType="client:ExampleBPEL4WS2YAWLRequestMessage"/>
```

```
        <variable name="outputVariable"
messageType="client:ExampleBPEL4WS2YAWLResponseMessage"/>
      </variables>
    <!-- ================================================================= -->
    <!-- ORCHESTRATION LOGIC--><!-- Set of activities coordinating the flow of messages across
the -->
    <!-- services integrated within this business process -->
    <!-- ================================================================= -->
      <sequence name="main"><!-- Receive input from requestor.
        Note: This maps to operation defined in ExampleBPEL4WS2YAWL.wsdl
        -->
        <receive name="receiveInput" partnerLink="client"
portType="client:ExampleBPEL4WS2YAWL" operation="process" variable="inputVariable"
createInstance="yes"/><!-- Generate reply to synchronous request -->
        <flow name="Flow_1">
          <links>
            <link name="AGreaterThanB"/>
            <link name="BGreaterThanA"/>
          </links>
          <switch name="Switch_1">
             <case condition="bpws:getVariableData('inputVariable','first') &lt;=
bpws:getVariableData('inputVariable','second')">
              <assign name="Inversion">
                <source linkName="BGreaterThanA" transitionCondition="true()"/>
                <copy>
                  <from variable="inputVariable" part="second"/>
                  <to variable="temporaryVar" part="first"/>
                </copy>
                <copy>
                  <from variable="inputVariable" part="first"/>
                  <to variable="temporaryVar" part="second"/>
                </copy>
              </assign>
            </case>
            <otherwise>
              <empty name="NoInversion">
                <source linkName="AGreaterThanB"/>
              </empty>
            </otherwise>
          </switch>
          <sequence name="Sequence_1">
            <assign name="CopyInversion" joinCondition="bpws:getLinkStatus('BGreaterThanA') and
not bpws:getLinkStatus('AGreaterThanB')" suppressJoinFailure=”yes”>
              <target linkName="BGreaterThanA"/>
              <target linkName="AGreaterThanB"/>
              <copy>
                <from variable="temporaryVar" part="first"/>
                <to variable="inputVariable" part="first"/>
              </copy>
              <copy>
                <from variable="temporaryVar" part="second"/>
                <to variable="inputVariable" part="second"/>
              </copy>
            </assign>
            <assign name="FromInputToOutput">
              <copy>
                <from variable="inputVariable" part="first"/>
                <to variable="outputVariable" part="greater"/>
              </copy>
              <copy>
                <from variable="inputVariable" part="second"/>
                <to variable="outputVariable" part="lesser"/>
```

```
          </copy>
         </assign>
        </sequence>
       </flow>
       <reply name="replyOutput" partnerLink="client" portType="client:ExampleBPEL4WS2YAWL"
operation="process" variable="outputVariable"/>
     </sequence>
    </process>
```

Tale esempio riguarda un processo che esegue le seguenti *attività*:

- Riceve (tramite l'attività il cui nome è **receiveInput**) dal cliente del servizio un messaggio composto da due valori, e lo salva nella sua variabile **inputVariable**.
- Manda in esecuzione concorrente, tramite l'*attività* chiamata **Flow_1**, le seguenti *attività*:

    - Una switch, **Switch_1**, che controlla i due valori: se il primo è minore o uguale al secondo, li copia, tramite l'*attività* **Inversion**, in posizione invertita nella variabile **temporaryVar**, anche questa composta da due valori, e manda il valore **true** attraverso il link **BGreaterThanA**; se il primo valore invece è già maggiore del secondo, esegue l'*attività* **NoInversion**, una **empty**, che manda il valore **true** sul link **AGreaterThanB**.
    - Una sequenza, **Sequence_1**, composta da due *attività*: la prima che copia il valore di **temporaryVar** nella variabile di input **inputVariable**, ma solo se i valori dei link **BGreaterThanA** e **AGreaterThanB** sono tali da dimostrare che è stata eseguita l'*attività* **Inversion**, altrimenti questa viene saltata (e non genera errori, visto che il suo attributo **suppressJoinFailure** ha come valore **yes**); poi c'è l'*attività* **FromInputToOutput** che copia la variabile di input **inputVariable** in quella di output **outputVariable**.

- Infine restituisce indietro al cliente del servizio il valore dell' **outputVariable** che conterrà i due valori mandati ordinati per dimensione decrescente.

L'esempio, in sintesi, mostra un servizio che, ricevuti due valori, li ordina in maniera decrescente e li rispedisce al mittente. Questo servizio è rappresentato graficamente

nella seguente figura:

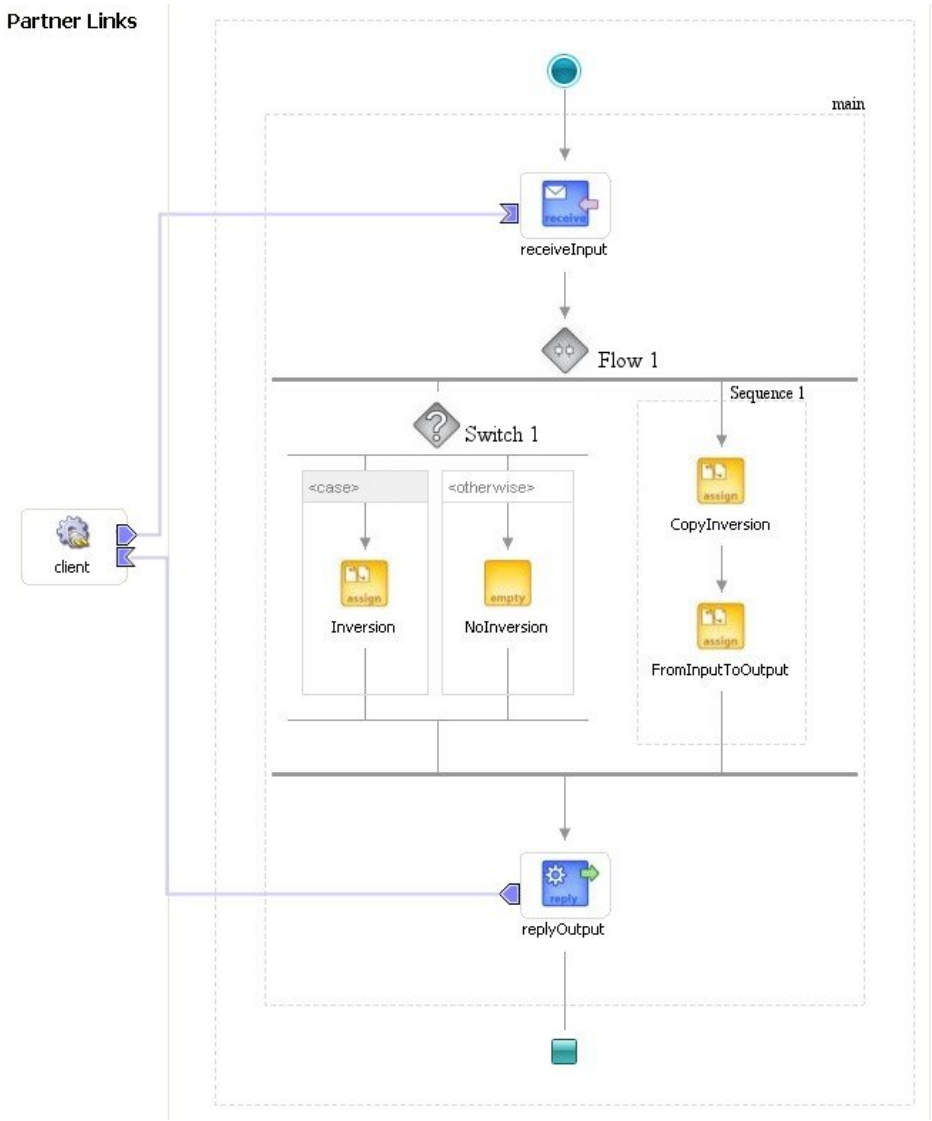


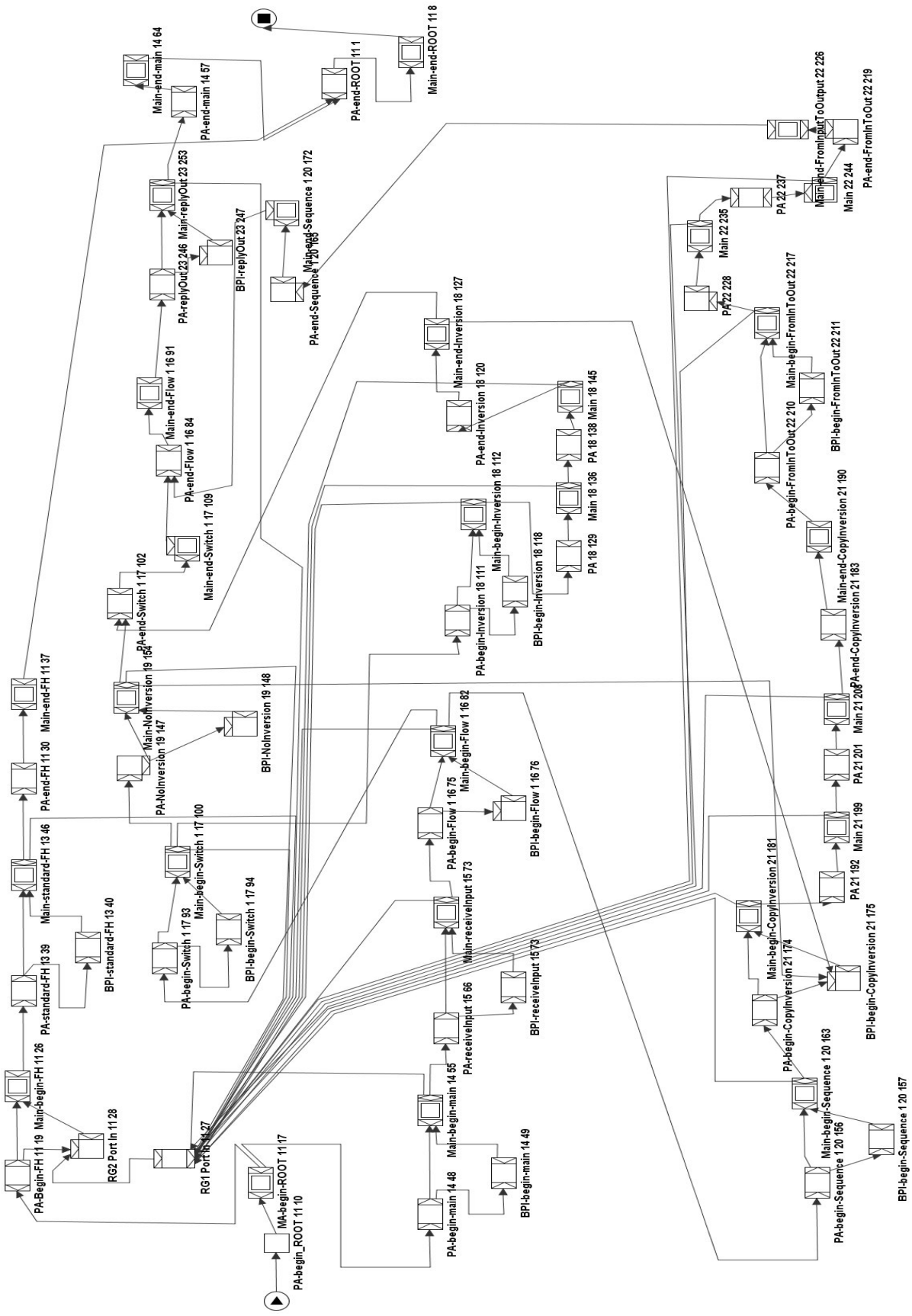
Figura 4.46: schema del processo BPEL4WS dell'esempio

Come abbiamo detto non è stato possibile mostrare il codice prodotto per intero. Tuttavia è stata creata a mano una sua rappresentazione grafica.

Nonostante non sia stato possibile evidenziare tutte le sottoreti (quelle che costituiscono i vari **Main Task**), lo schema è interessante in quanto, oltre a dimostrare un esatto funzionamento del programma sviluppato, sottolineando quindi che è possibile ottenere una traduzione automatica di un processo BPEL4WS che rispecchi la metodologia di traduzione discussa in questa tesi, mostra anche il risultato

dell'applicazione teorica di tutta la metodologia di traduzione discussa in *4.1*.

Qui di seguito viene mostrata l'immagine del workflow creata usando l'editor YAWL.


Main-end-main 14 64
PA-end-main 14 57
PA-end-ROOT 11 1
Main-end-ROOT 11 8
Main-replyOut 23 253
BPI-replyOut 23 247
PA-replyOut 23 246
Main-end-Sequence 1 20 172
PA-end-Sequence 1 20 168
Main-end-Flow 1 16 91
PA-end-Flow 1 16 84
Main-end-Switch 1 17 109
PA-end-Switch 1 17 102
Main-NoInversion 19 147
BPI-NoInversion 19 148
PA-NoInversion 19 154
Main-end-FH 11 37
PA-end-FH 11 30
Main-standard-FH 13 46
BPI-standard-FH 13 40
PA-standard-FH 13 39
Main-begin-FH 11 26
RG2 Port In 11 28
RG1 Port In 11 27
PA-Begin-FH 11 19
MA-begin-ROOT 11 17
PA-begin_ROOT 11 10
Main-begin-Switch 1 17 100
BPI-begin-Switch 1 17 94
PA-begin-Switch 1 17 93
Main-begin-Flow 1 16 82
BPI-begin-Flow 1 16 76
PA-begin-Flow 1 16 75
Main-receiveInput 15 73
BPI-receiveInput 15 73
PA-receiveInput 15 66
Main-begin-main 14 55
BPI-begin-main 14 49
PA-begin-main 14 48
Main-end-inversion 18 127
PA-end-inversion 18 120
Main 18 145
PA 18 138
Main 18 136
PA 18 129
Main-begin-Inversion 18 112
BPI-begin-inversion 18 118
PA-begin-Inversion 18 111
Main-end-CopyInversion 21 190
Main-end-CopyInversion 21 183
Main 21 208
PA 21 201
Main 21 199
PA 21 192
Main-begin-CopyInversion 21 181
BPI-begin-CopyInversion 21 175
Main-begin-CopyInversion 21 174
Main-begin-Sequence 1 20 163
BPI-begin-Sequence 1 20 157
Main-begin-Sequence 1 20 156
PA-begin-Sequence 1 20 155
PA-begin-FromInToOut 22 217
BPI-begin-FromInToOut 22 211
Main-begin-FromInToOut 22 210
PA 22 228
Main 22 235
PA 22 237
Main-end-FromInputToOutput 22 226
Main 22 244
PA-end-FromInToOut 22 219

Nonostante non sia possibile vedere la gestione dei dati è interessante vedere che viene rispettata la struttura dei pattern, che è facile riscontrare nelle terne di task (quelli che iniziano per **PA** sono i **Parental Advisory**, quelli che iniziano con **BPI** sono i **Blue Port In** e quelli che iniziano con **Main** sono i **Main Task**).

Il traduttore assegna un nome ad ogni task formato da una parte iniziale che indica quale parte del pattern rappresenta (**PA**, **BPI** o **Main**), un'altra parte del nome che indica se è il pattern iniziale (**begin**) o finale (**end**), il nome dell'*attività* BPEL4WS che il pattern rappresenta, e due valori: il primo è l'**_oid**, di cui abbiamo già parlato nel paragrafo *4.2.1.1*, ed è un identificatore del pattern (per fare un esempio basandoci sul workflow della figura sopra, l'*attività* **Sequence_1** ha un **_oid** pari a '20', infatti se guardiamo il pattern iniziale e quello finale, entrambi hanno quel numero; da notare che esisterà quindi una variabile nel workflow chiamata '**Prnt_skp_20**' che è associata a questa sequenza e che indica alle *attività* annidate in questa se devono essere saltate perché l'attività composita che le contiene, che è appunto la **Sequence_1**, deve essere saltata), il secondo è un valore unico per ogni task, e fa sì che ogni task abbia una decomposizione unica.

Capitolo 5 - Conclusioni

In questa tesi è stata progettata una metodologia di traduzione che permettesse di rappresentare un processo BPEL4WS attraverso un workflow YAWL. Tale metodologia è basata su pattern, ovvero 'blocchi' di workflow YAWL che rappresentano parte dei costrutti di BPEL4WS, quindi parte di una attività, o di un gestore di eventi o di fallimenti, o altro.

Tali pattern sono stati concepiti in modo da modellare tutte le possibili esecuzioni delle parti di processo BPEL4WS che rappresentano, e portano ad una traduzione che può essere concettualmente divisa in due parti: una prima parte in cui vengono create tutte le istanze di tali pattern utili a rappresentare tutto il processo, e una seconda parte in cui tali istanze vengono collegate assieme in base al significato che hanno, come spiegato nel paragrafo *4.1*.

Dopo aver elaborato questa metodologia è stato sviluppato un software, BPEL2YAWL, scritto in Java e descritto nel paragrafo *4.2*, in grado di compiere questa traduzione in maniera automatica, tale cioè da fare il parsing di un documento BPEL4WS, crearsene una rappresentazione mediante gli oggetti di una delle librerie di cui è composto tale software, istanziare i pattern da utilizzare per rappresentare il processo e collegarli assieme seguendo la metodologia elaborata.

La traduzione da BPEL4WS a YAWL proposta in questa tesi presenta diversi vantaggi rispetto ai lavori analoghi che costituiscono lo stato dell'arte per quanto riguarda la rappresentazione del linguaggio BPEL4WS con tecnologie di formalizzazione diverse. Tali vantaggi sono riassumibili nei diversi punti:

1. il traduttore BPEL2YAWL è l'unico attualmente esistente che riguardi YAWL come linguaggio di specifica, sono stati trattati *tutti* gli aspetti della specifica BPEL4WS ed è stato usato un approccio composizionale basato su pattern.
2. L'implementazione Java permette di dimostrare che tale traduzione può essere

veramente automatizzata.

3. La traduzione in sé permette di effettuare verifiche sul processo tradotto, e migliora quindi il risultato di una eventuale aggregazione.

Il primo e più importante vantaggio è chiaramente legato alla metodologia di traduzione. Avere usato YAWL come linguaggio di workflow rende questa tesi particolarmente interessante anche a confronto dei lavori basati sulle reti di Petri (tali lavori, ricordiamolo, sono quelli che sono risultati più completi, nel senso che riguardano un maggior numero di aspetti della specifica BPEL4WS). Questo perché, come discusso nel paragrafo *3.1*, YAWL risulta essere "discendente" delle reti di Petri, come se fosse una versione perfezionata di queste ultime, ma espressivamente più potenti e di più facile gestione.

Così tramite YAWL è stato possibile non solo gestire il comportamento del servizio a livello di esecuzione, ma anche i dati, nonostante per il momento il lavoro si sia limitato a gestire in maniera minimale tale aspetto, e solo per quanto riguarda le espressioni XPath che alteravano in maniera diretta il comportamento del workflow (come per esempio le **`joinCondition`** e i valori degli status dei *link* o le altre espressioni booleane).

Tuttavia il maggior potere espressivo di YAWL permette anche una più semplice definizione di alcuni comportamenti di BPEL4WS: si pensi ad esempio alla terminazione delle *attività* che in YAWL si esegue mandando semplicemente il flusso al pattern finale che esegue una cancellazione di token su tutto il processo, mentre, in mancanza del pattern di cancellazione (come spiegato nel paragrafo *3.2* pattern 20), che non è presente in linguaggi come quelli basati sulle reti di Petri, si è costretti a complicare tutti i task per permettere un controllo sulla possibile terminazione (si veda a tal proposito la soluzione trovata in [8]).

YAWL è una tecnologia nuova e in continua espansione, con sempre nuove funzioni, e questo comporta un sempre maggior interesse da parte sia della comunità scientifica che da parte di quella industriale. Infatti un altro vantaggio di questo lavoro rispetto ad altri è la sua possibilità di utilizzo: esistono già lavori basati sul linguaggio YAWL che

riguardano il campo dei Web Services, come ad esempio quello mostrato in [10].

Era quindi molto interessante sviluppare una metodologia di traduzione, come quella vista in questa tesi, che permettesse la gestione di tutte le particolarità del linguaggio BPEL4WS. Questo è un risultato importante che difficilmente è stato raggiunto in altri lavori. I lavori svolti precedentemente a questa tesi e che hanno usato tecniche di formalizzazione quali le macchine a stati astratti (ASM da *Abstract State Machine*), le macchine a stati finiti (FSM da *Finite State Machines*) o ancora le algebre di processi (PA da *Process Algebras*) non hanno trattato alcune particolarità del linguaggio quali possono essere la gestione dei *link* oppure i gestori di eventi e errori. I lavori sviluppati tramite le reti di Petri sono risultati più completi, ma abbiamo già detto che le potenzialità espressive di YAWL rendono il lavoro di questa tesi molto interessante.

Da notare che è indispensabile, sempre nell'ottica dell'utilizzo di questo traduttore come componente di software per l'aggregazione di servizi Web, che tutti gli aspetti del linguaggio BPEL4WS vengano rappresentati in maniera adeguata. Un lavoro che non tratti uno di questi elementi rischia di non permettere una aggregazione automatica esatta.

E sempre in vista di un possibile aggregatore automatico la particolarità di avere una traduzione basata su pattern rende possibile lo sviluppo di un futuro traduttore inverso, da YAWL a BPEL4WS, che funzioni basandosi proprio sull'interpretazione inversa dei pattern, utile per ottenere una descrizione BPEL4WS del processo composto da servizi tradotti precedentemente in YAWL. Questo permetterebbe anche di avere software automatici per l'aggregazione di servizi descritti in linguaggi diversi e che usino YAWL come lingua franca.

Inoltre, come è stato già discusso nell'introduzione a questa tesi, BPEL4WS è privo di una semantica formale, e questo limita molto le possibilità di dimostrazione di talune proprietà, possibilità che si avrebbero invece con una traduzione in un linguaggio come YAWL. L'importanza di questa tesi è anche ricercabile nella possibilità di generare un workflow YAWL che rappresenti in tutti i suoi aspetti un processo BPEL4WS in modo da poter utilizzare i tool di analisi semantica che vengono costantemente espansi e perfezionati con lo sviluppo di versioni sempre più evolute dell'editor e dell'engine di

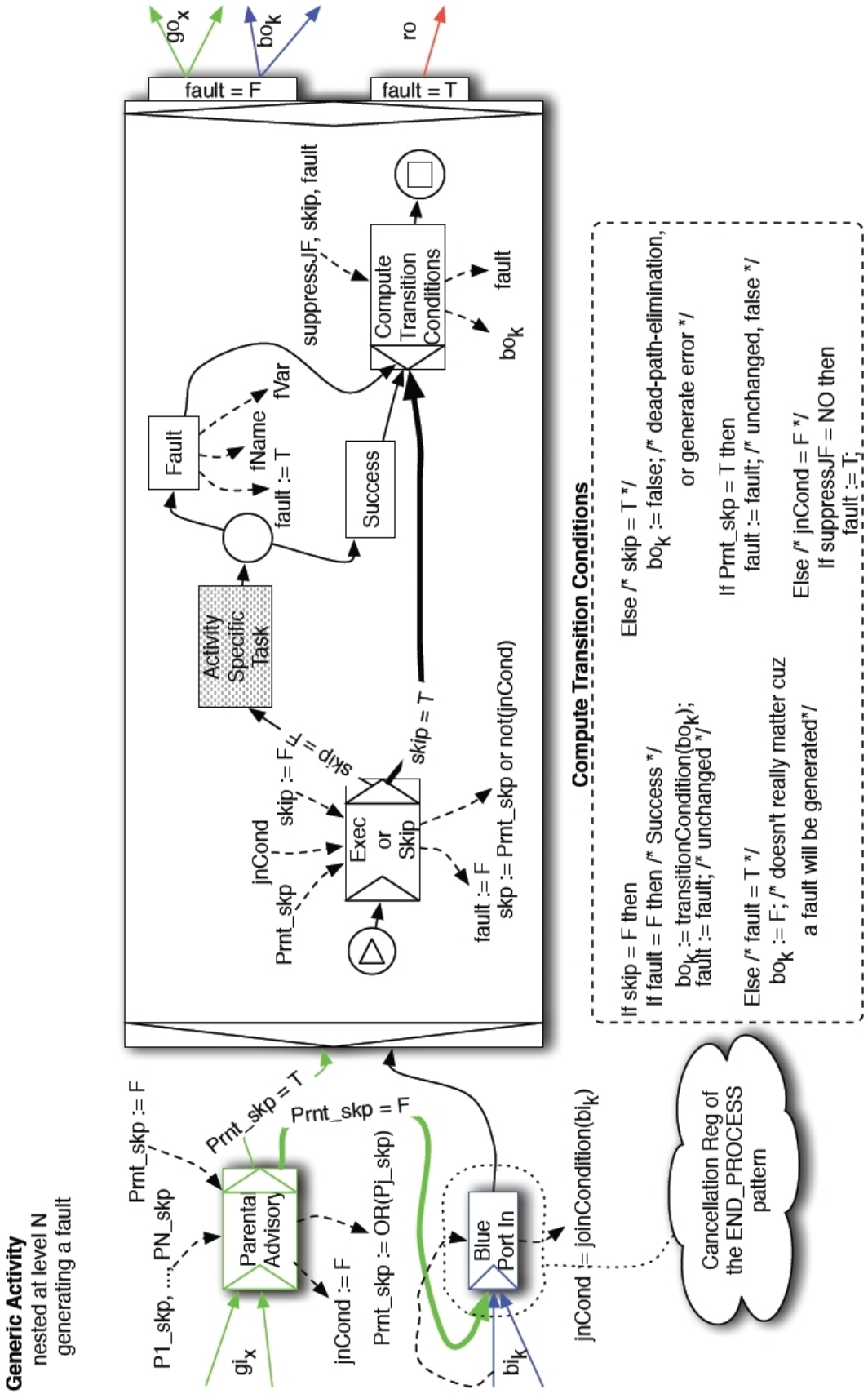
YAWL.

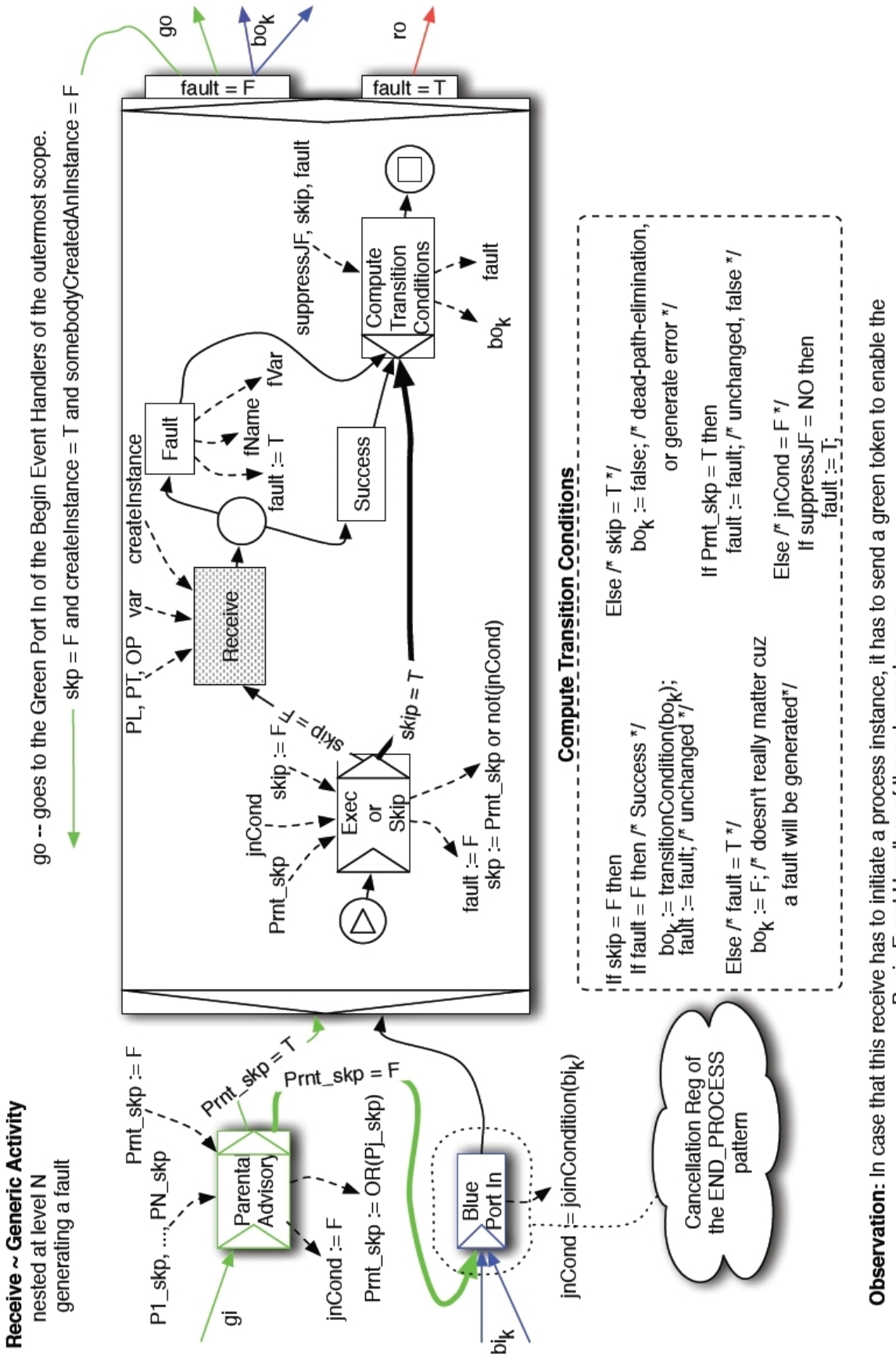
Sempre per considerare ulteriori sviluppi futuri, potrebbe essere ulteriormente espansa la gestione dei dati di BPEL4WS nel workflow YAWL. La possibilità che mette a disposizione YAWL di trattare variabili complesse definite nel proprio **`schema`**, come visto per il tipo di dato complesso **`Geek`** discusso nel paragrafo *3.6*, potrebbe essere sfruttata per importare i tipi di dati descritti in XMLSchema nei documenti WSDL ed avere così la possibilità di simulare anche le variabili dei messaggi nella maniera esatta in cui vengono usate nel processo BPEL4WS.

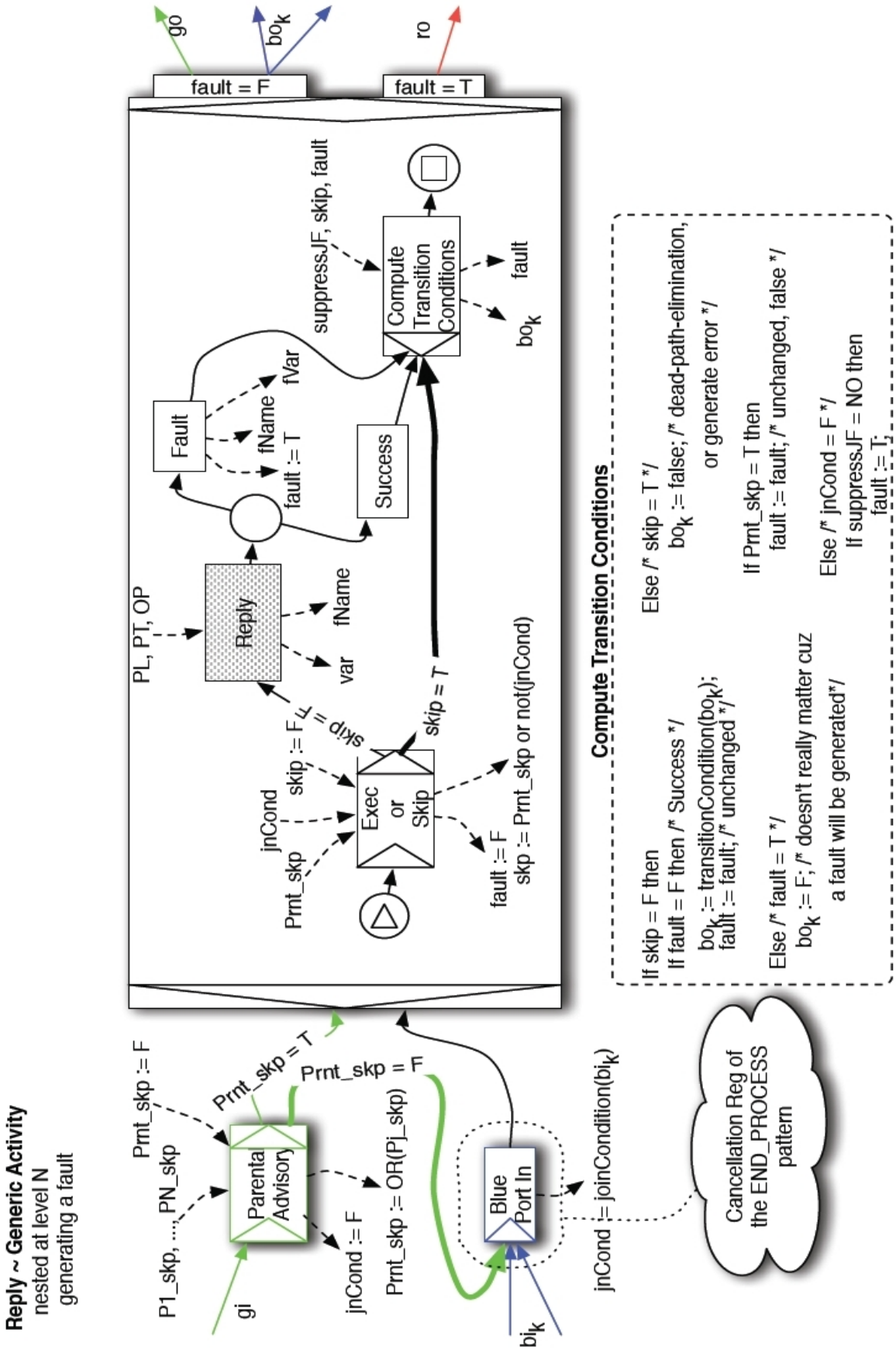
Infine un ultimo sviluppo futuro di questo lavoro potrebbe essere quello di implementare la gestione di default della compensazione (che si ha nel caso di definizione di una *attività* **`compensate`** priva del campo **`scope`** o nel caso di mancanza del gestore della compensazione), che prevede, come detto nel paragrafo *2.7.5*, l'invocazione dei gestori della *compensazione* per gli **`scope`** interni a quello riferito nell'ordine inverso di completamento di questi **`scope`**.. Per fare questo basterebbe utilizzare come struttura una pila *(stack)*.

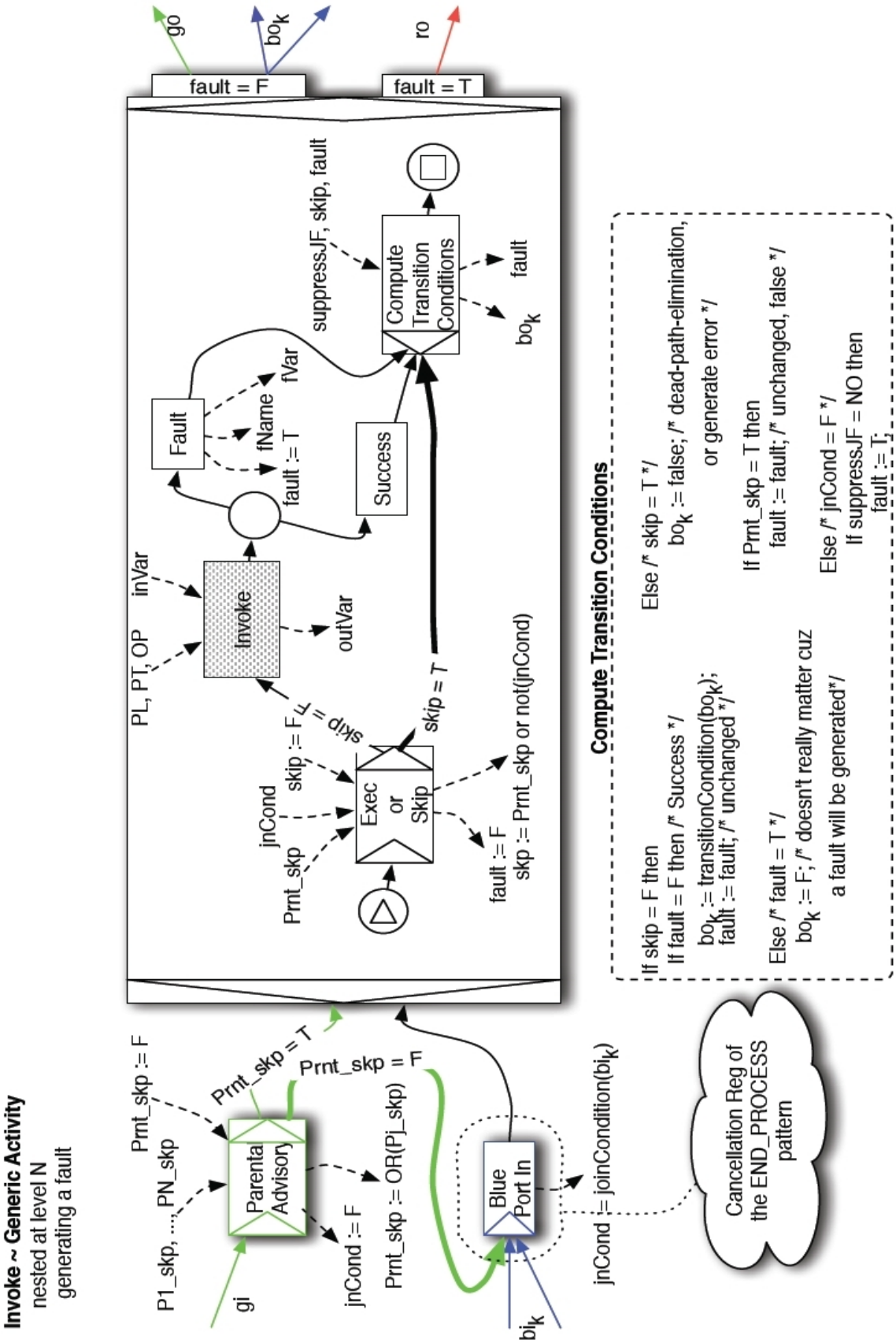
Per permettere lo sviluppo di questo lavoro il codice è stato pubblicato come progetto su SourceForge (https://sourceforge.net/projects/bpel2yawl/).

Appendice A : Pattern di Traduzione

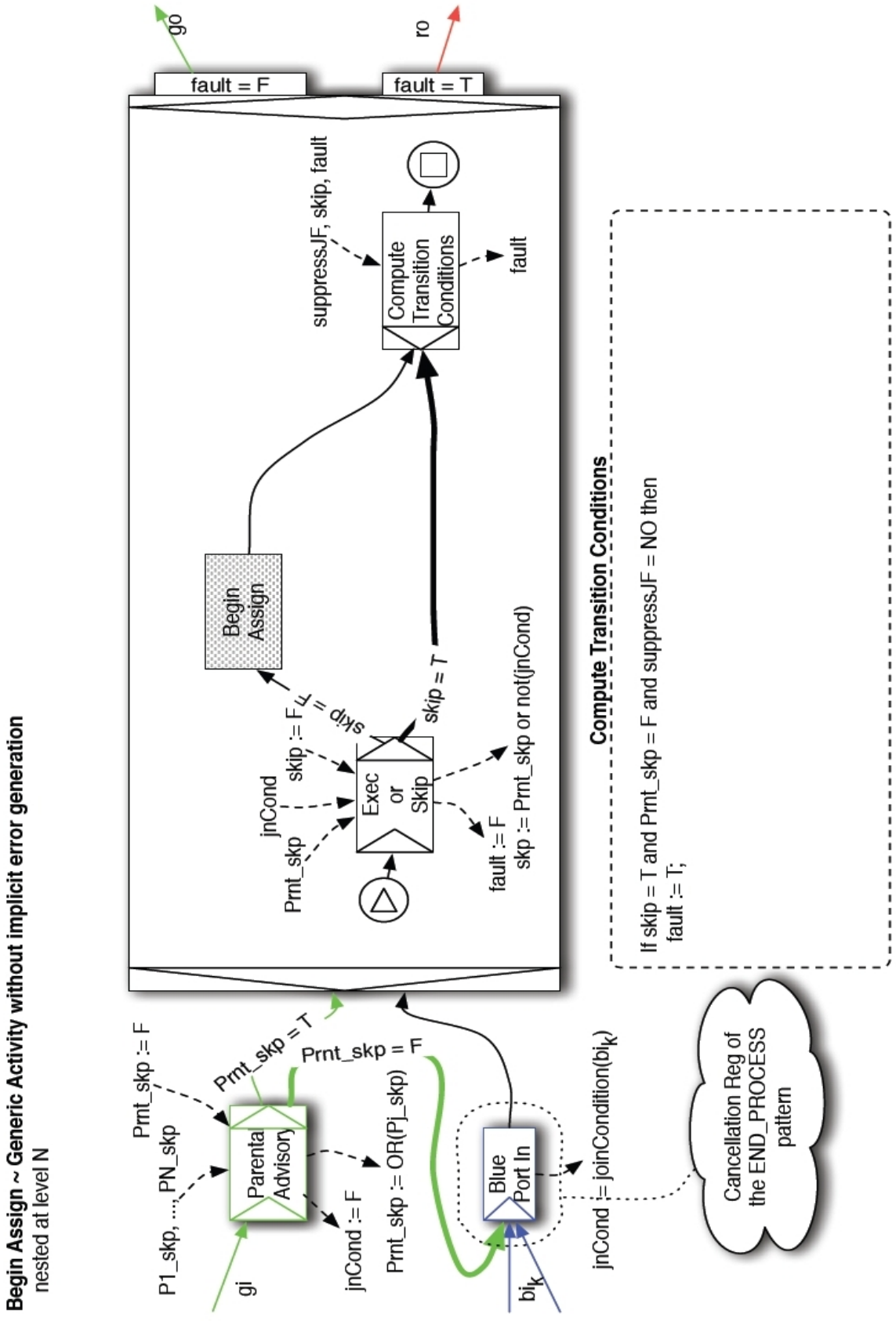

Generic Activity
nested at level N
generating a fault
P1_skp, ..., PN_skp
Prnt_skp := F
gi_x
Parental Advisory
jnCond := F
Prnt_skp := OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
bi_k
Blue Port In
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
go_x
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Fault
fName
fVar
fault := T
Success
Activity Specific Task
skip := F
skip = T
jnCond
Prnt_skp
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
In terms of YAWL mappings, we could write:
1. bo_k := NOT(skip) AND transitionCondition(bo_k);
2. fault := fault OR (NOT jnCond AND suppressJF = NO)

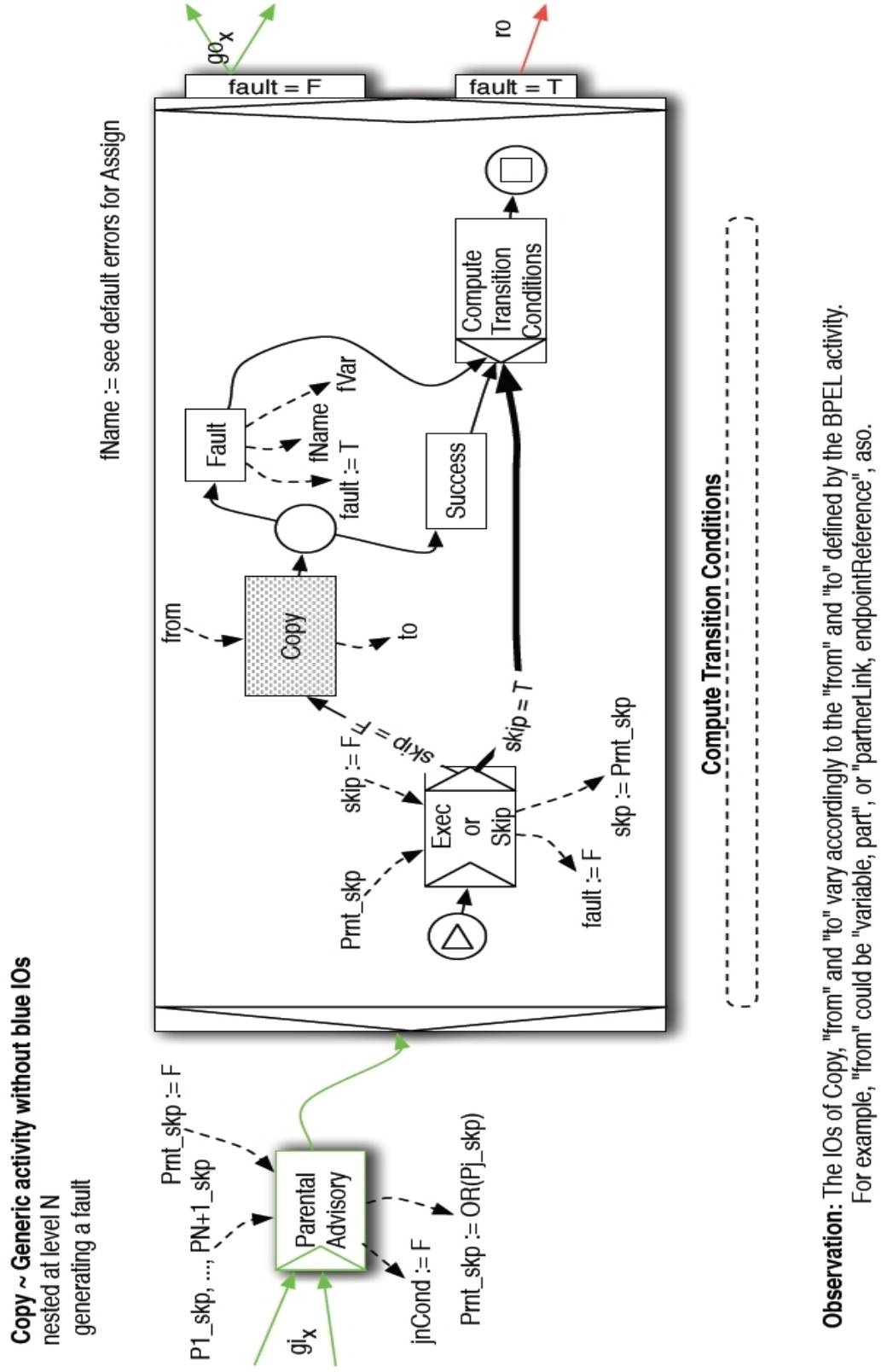

Receive ~ Generic Activity
nested at level N
generating a fault
go -- goes to the Green Port In of the Begin Event Handlers of the outermost scope.
skp = F and createInstance = T and somebodyCreatedAnInstance = F
go
bo_k
ro
fault = F
fault = T
PL, PT, OP
var
createInstance
Receive
Fault
fName
fVar
fault := T
Success
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
jnCond
Prnt_skp
skip := F
skip = T
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
Prnt_skp = T
Prnt_skp = F
jnCond := F
Prnt_skp := OR(Pj_skp)
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
Observation: In case that this receive has to initiate a process instance, it has to send a green token to enable the Begin Event Handlers of the outermost scope.

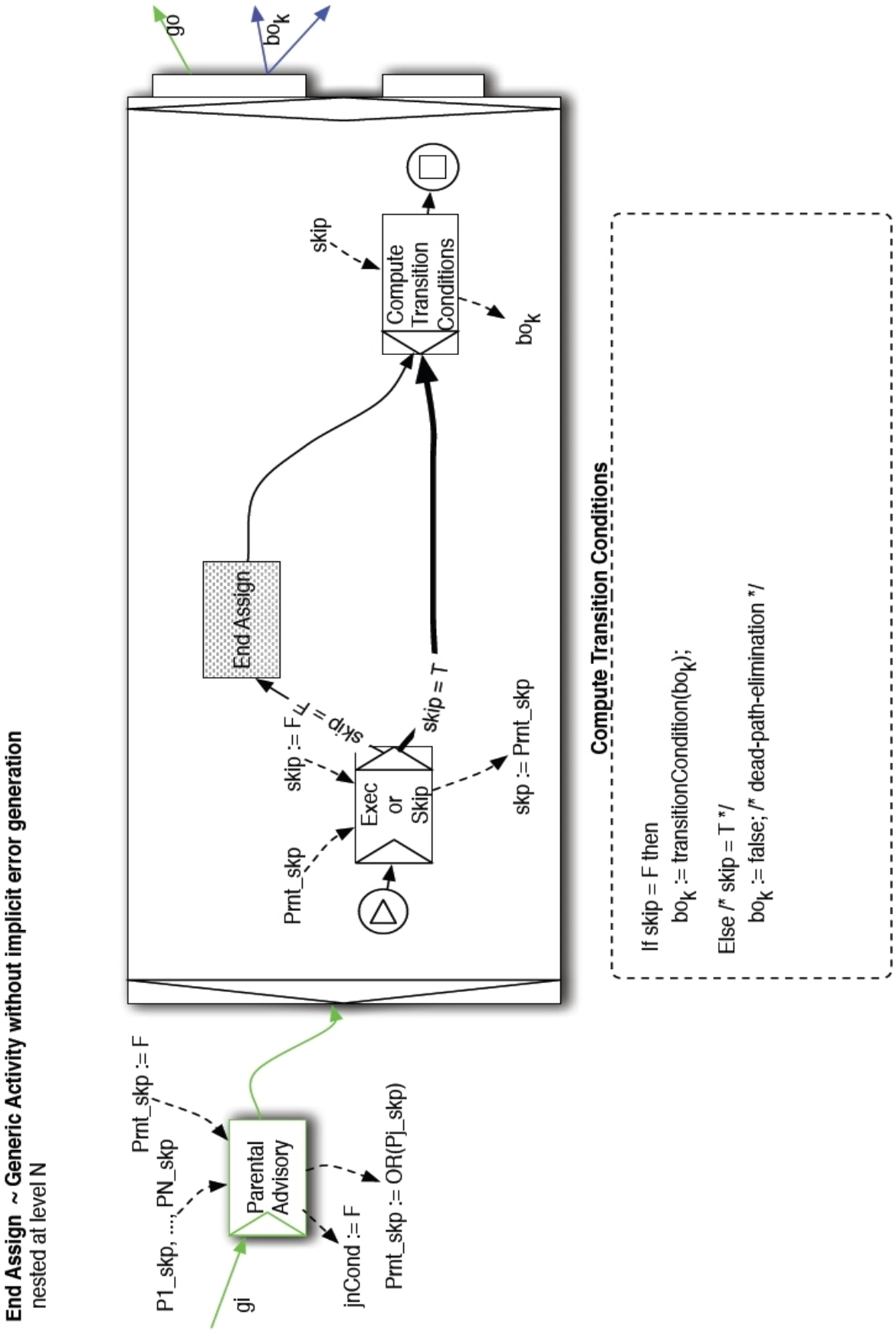

Reply ~ Generic Activity
nested at level N
generating a fault
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
jnCond := F
Prnt_skp := OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
go
bo_k
ro
fault = F
fault = T
PL, PT, OP
Reply
fName
var
skip = F,T
jnCond
Prnt_skp
skip := F
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Fault
fVar
fName
fault := T
Success
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;

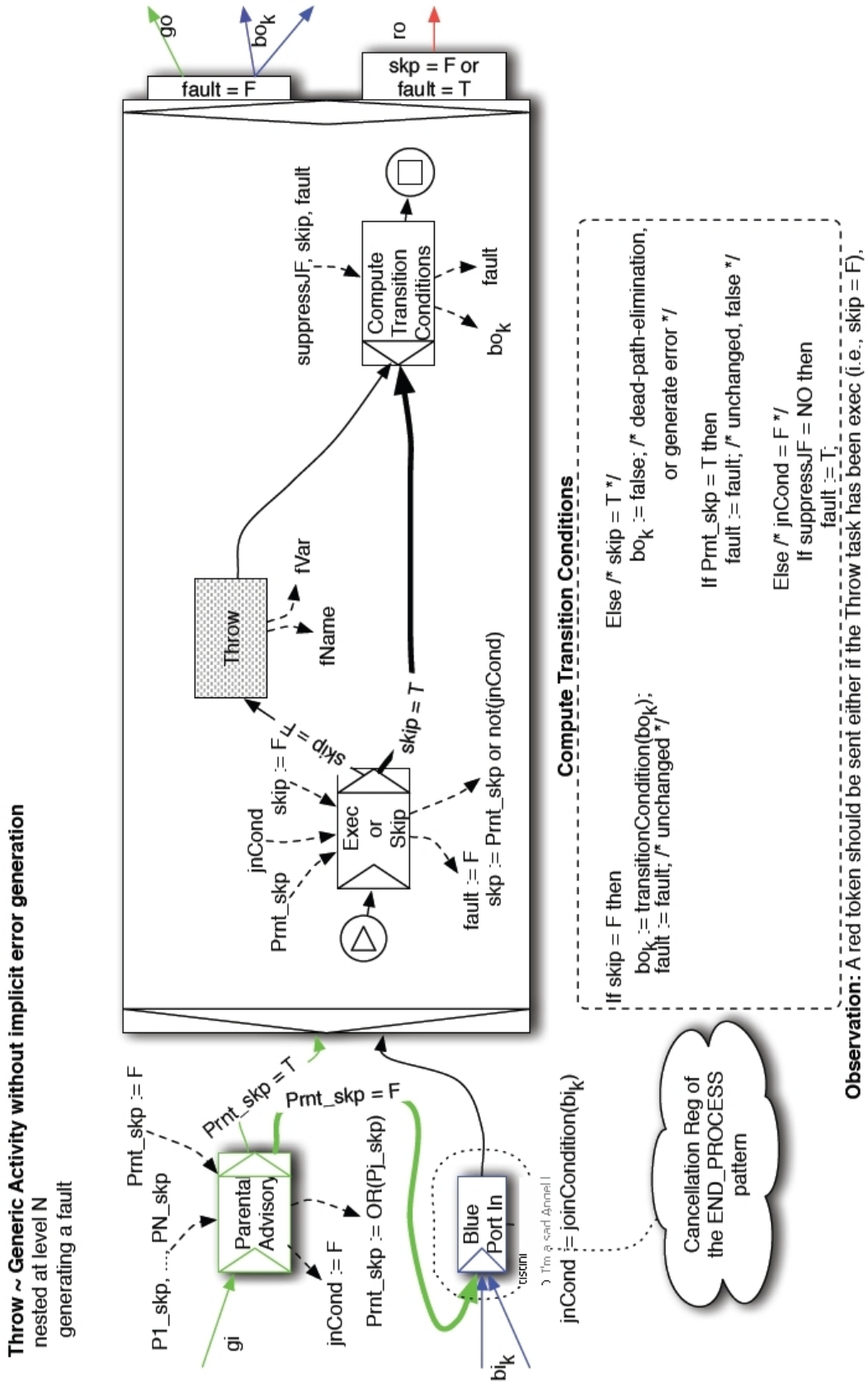

Invoke ~ Generic Activity
nested at level N
generating a fault
go
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Fault
fName
fVar
fault := T
Success
Invoke
inVar
PL, PT, OP
outVar
skip := F
skip = T
jnCond
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parenta Advisory
Prnt_skp = T
Prnt_skp = F
jnCond := F
Prnt_skp := OR(Pj_skp)
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern

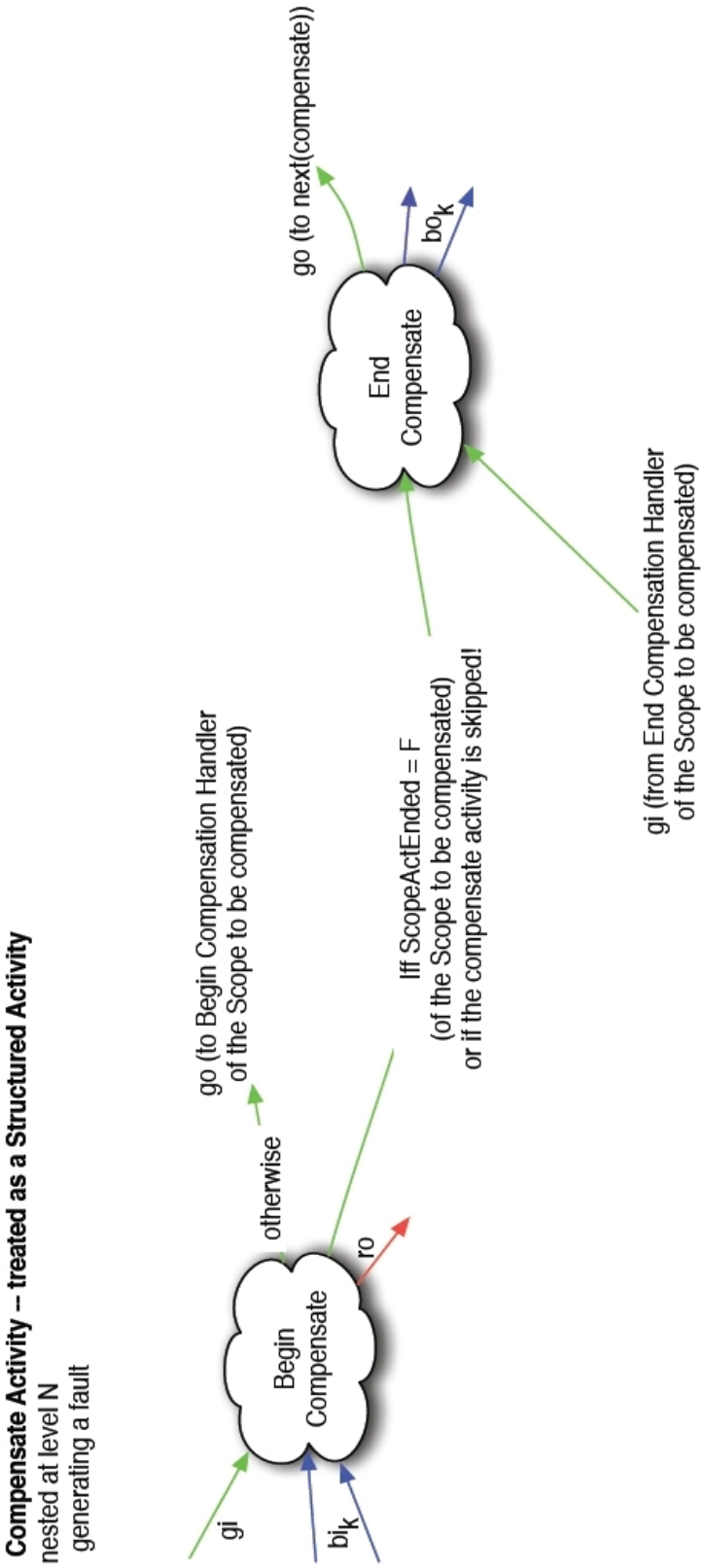
Assign Activity ~ Sequence Structured Activity
nested at level N
generating a fault
gi
bi_k
Begin Assign
go_x
gi_x
ro
Copy
go_x
gi_x
ro
Copy
go_x
gi_x
ro
End Assign
go
bo_k
where:
Begin Assign
=
Empty (without blue Os)
and
Copy
=
Empty (without blue IOs)
and
End Assign
=
Empty (without blue Is and red Os)

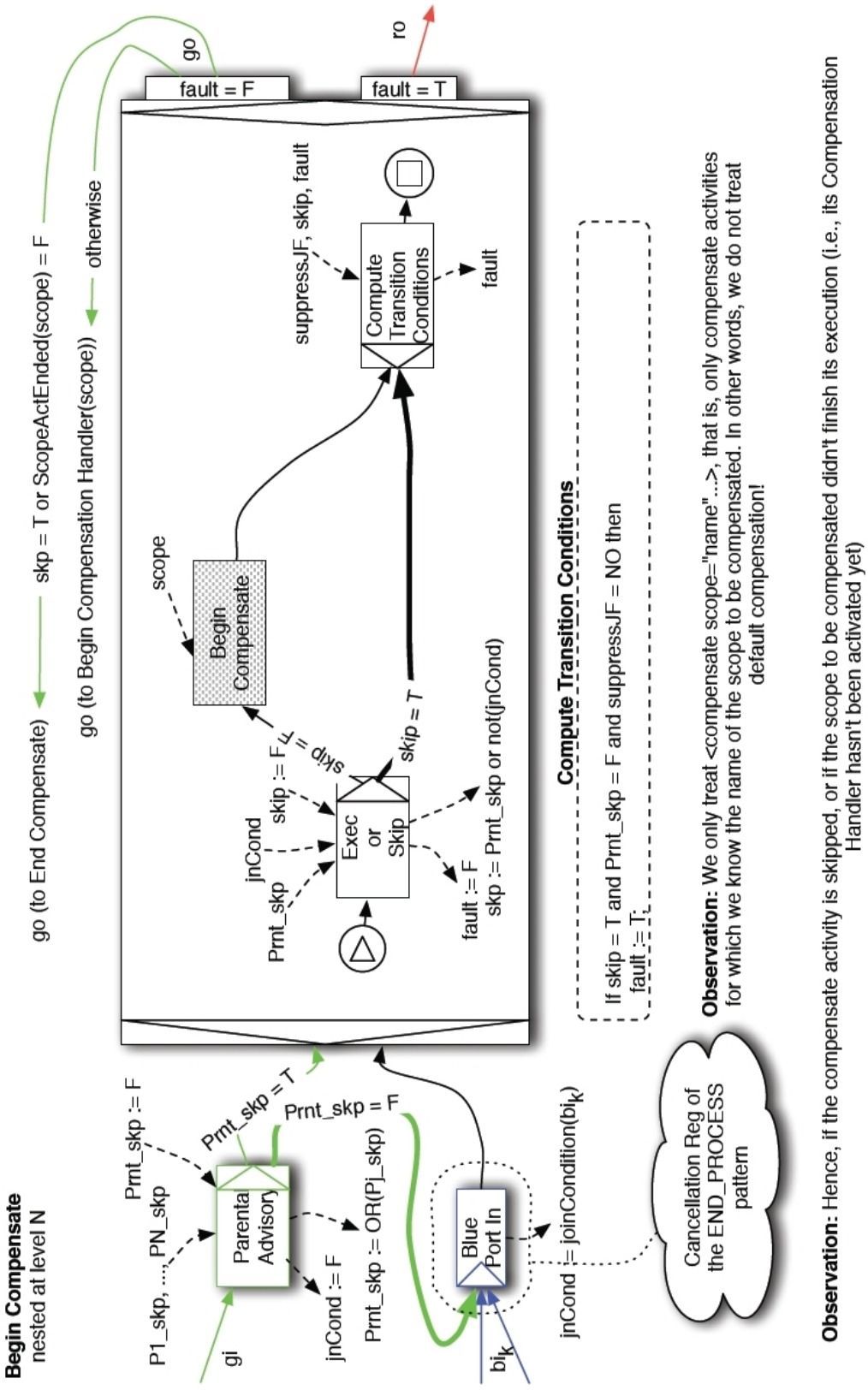

Begin Assign ~ Generic Activity without implicit error generation
nested at level N
go
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin Assign
skip := F
skip := T
jnCond
Prnt_skp
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bik
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern

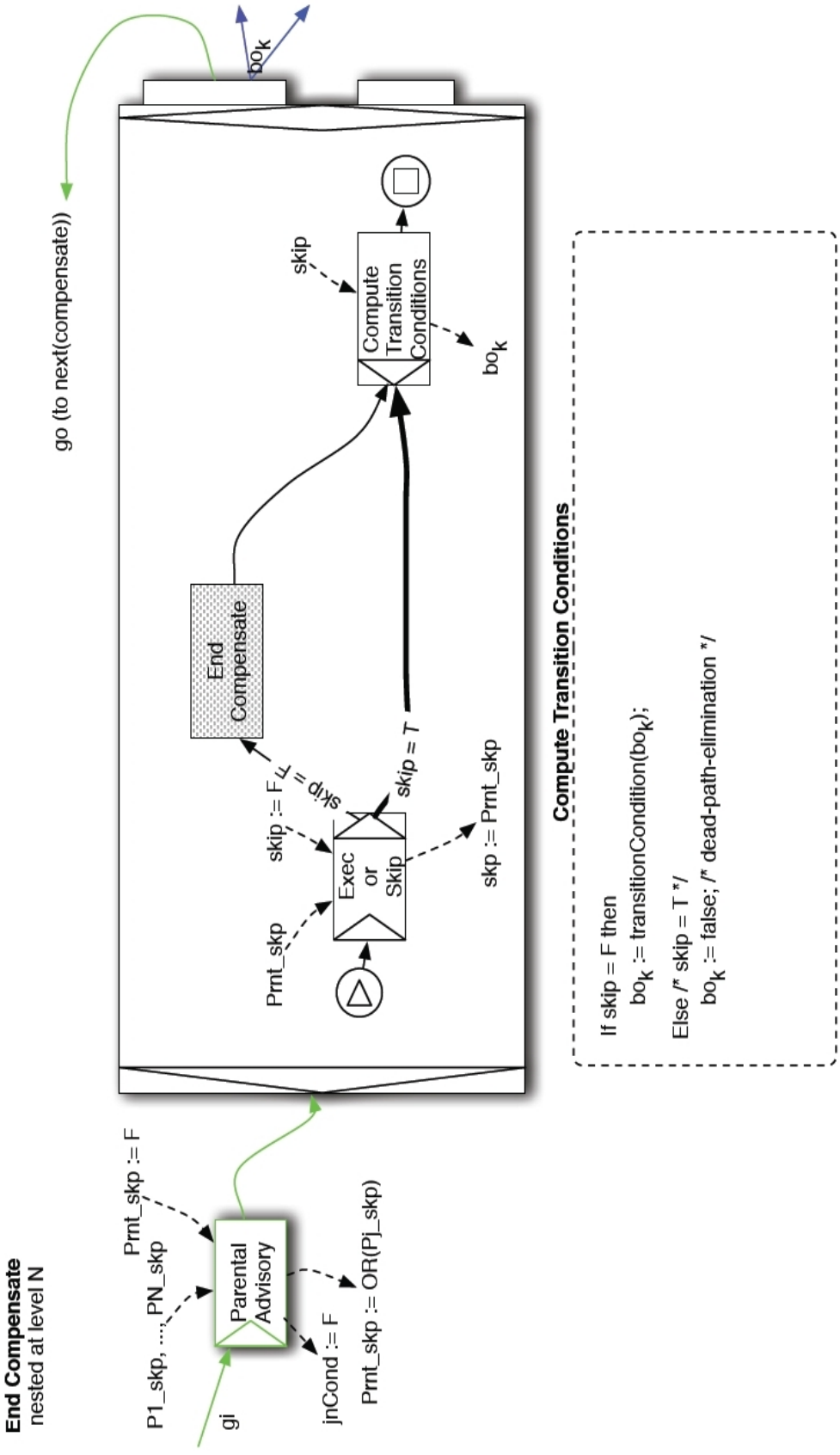

Copy ~ Generic activity without blue IOs
nested at level N
generating a fault
fName := see default errors for Assign
Pmt_skp := F
P1_skp, ..., PN+1_skp
Parental Advisory
gi_x
joinCond := F
Pmt_skp := OR(Pj_skp)
go_x
fault = F
fault = T
ro
from
Copy
to
Fault
fName
fVar
fault := T
Success
Compute Transition Conditions
skip := F
T = skip
skip = T
Pmt_skp
Exec or Skip
fault := F
skp := Pmt_skp
Compute Transition Conditions
Observation: The IOs of Copy, "from" and "to" vary accordingly to the "from" and "to" defined by the BPEL activity.
For example, "from" could be "variable, part", or "partnerLink, endpointReference", aso.

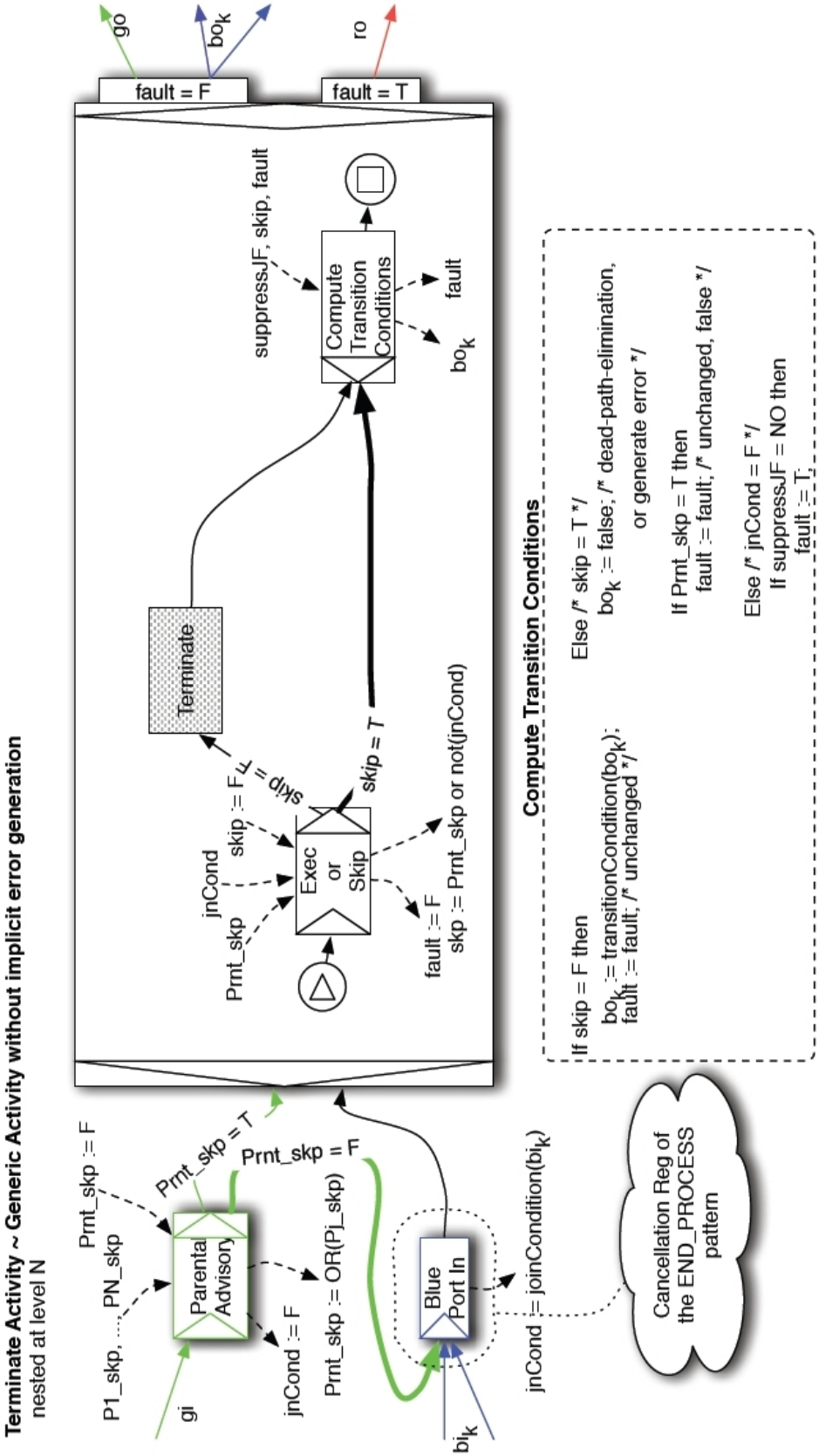

End Assign ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
jnCond := F
Prnt_skp := OR(Pj_skp)
go
bo_k
skip
Compute Transition Conditions
bo_k
End Assign
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

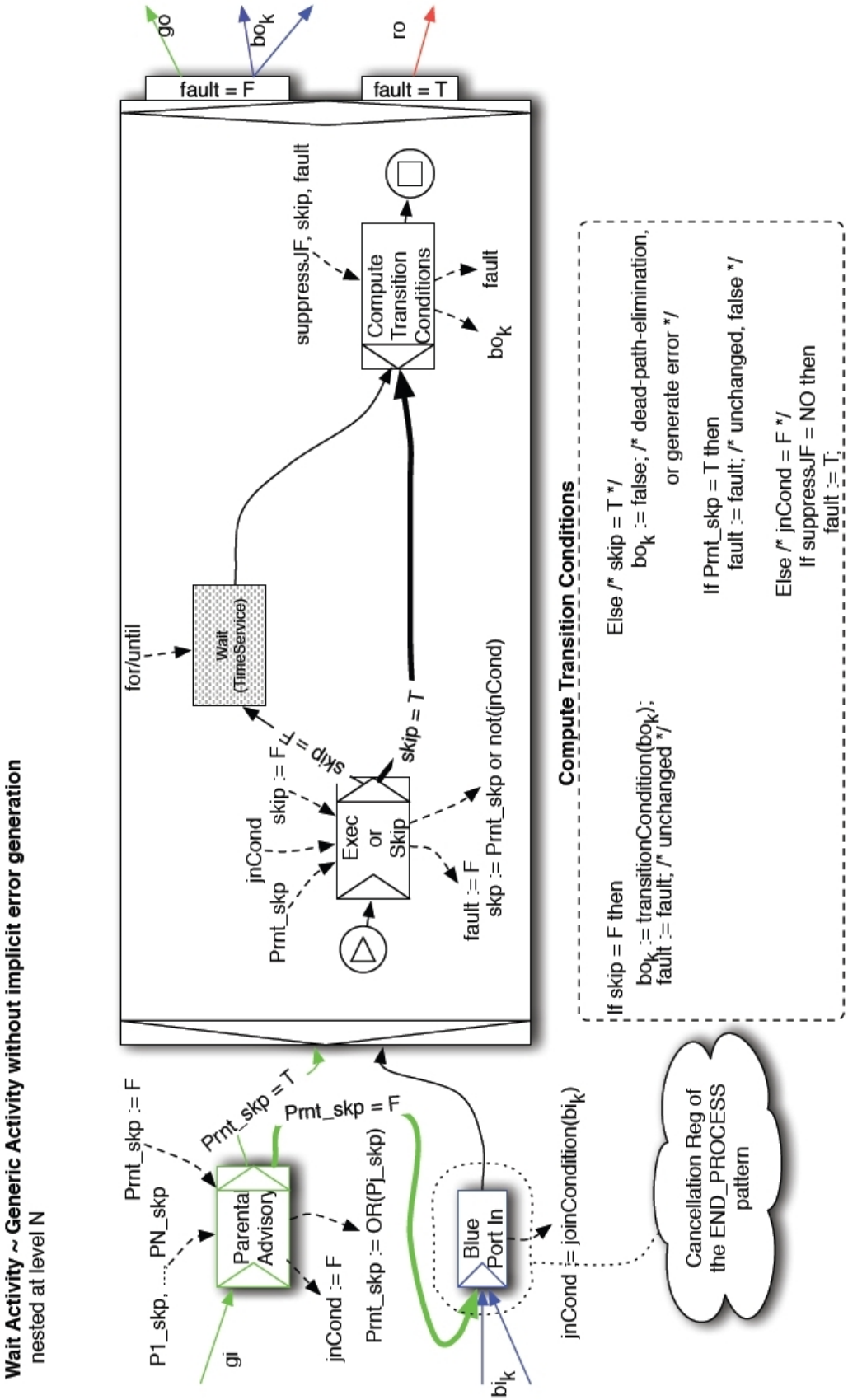

Throw ~ Generic Activity without implicit error generation
nested at level N
generating a fault
go
bo_k
ro
fault = F
skp = F or
fault = T
suppressJF, skip, fault
Compute
Transition
Conditions
fault
bo_k
Throw
fVar
fName
jnCond
Prnt_skp
skip := F
skip = T
Exec
or
Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Observation: A red token should be sent either if the Throw task has been exec (i.e., skip = F),
or if there is a joinFailure being generated (i.e., fault = T)!
Prnt_skp := F
P1_skp, ..., PN_skp
Parental
Advisory
gi
jnCond := F
Prnt_skp := T
Prnt_skp = F
Prnt_skp := OR(Pj_skp)
Blue
Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of
the END_PROCESS
pattern
Observation: If the pattern executes normally (i.e., skip = F), tokens of all colours will be sent!
Otherwise, what would be the sense of having a Throw as the source of a link?!?

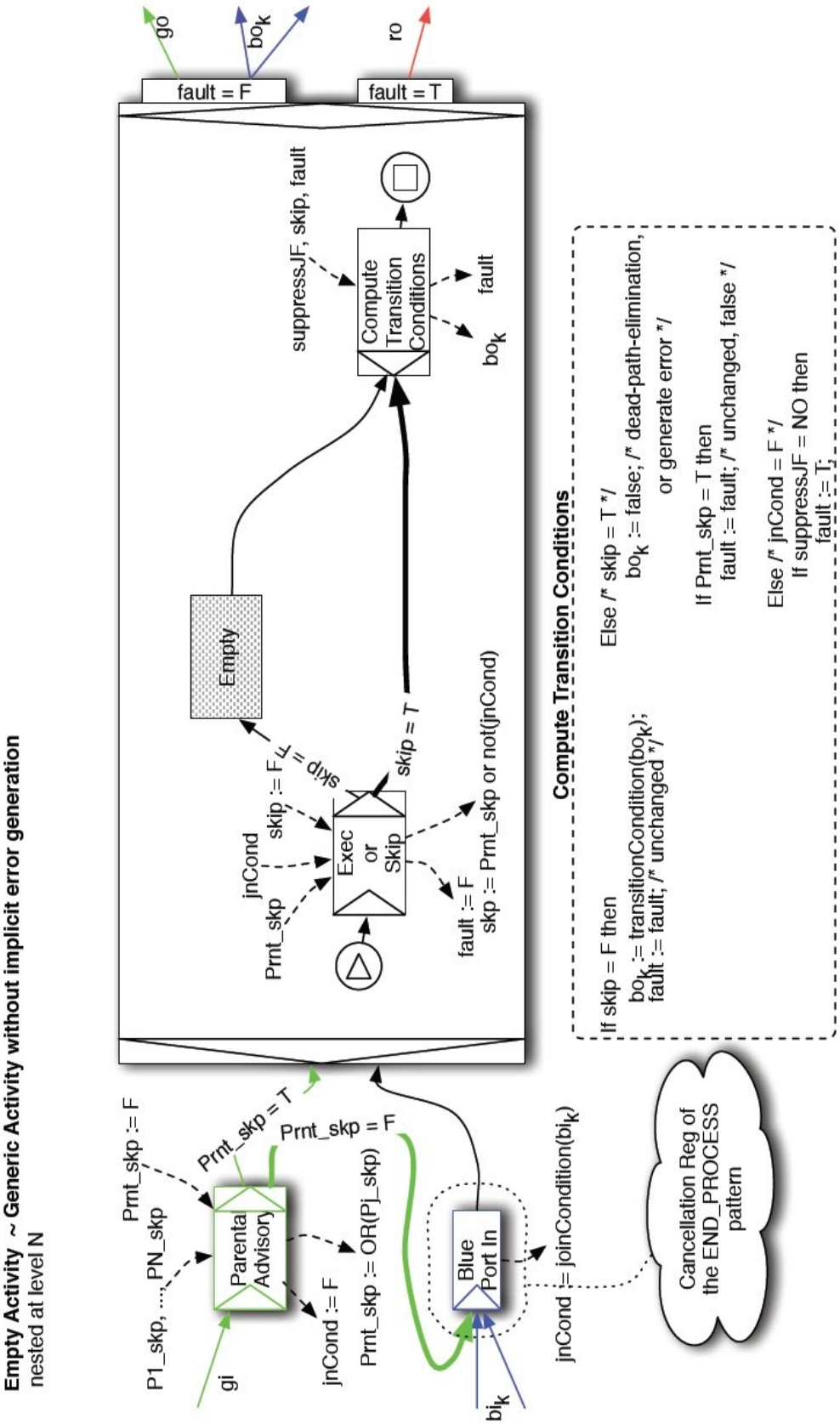

Compensate Activity – treated as a Structured Activity
nested at level N
generating a fault
gi
bi_k
Begin Compensate
ro
otherwise
go (to Begin Compensation Handler of the Scope to be compensated)
Iff ScopeActEnded = F
(of the Scope to be compensated)
or if the compensate activity is skipped!
End Compensate
bo_k
go (to next(compensate))
gi (from End Compensation Handler of the Scope to be compensated)


Begin Compensate
nested at level N
go (to End Compensate)
skp = T or ScopeActEnded(scope) = F
go (to Begin Compensation Handler(scope))
otherwise
go
ro
fault = F
fault = T
scope
Begin Compensate
suppressJF, skip, fault
Compute Transition Conditions
fault
jnCond
Prnt_skp
skip := F
skip := T
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Observation: We only treat <compensate scope="name"...>, that is, only compensate activities for which we know the name of the scope to be compensated. In other words, we do not treat default compensation!
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := T
Prnt_skp = F
Prnt_skp := OR(Pj_skp)
Blue Port In
bik
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern
Observation: Hence, if the compensate activity is skipped, or if the scope to be compensated didn't finish its execution (i.e., its Compensation Handler hasn't been activated yet)

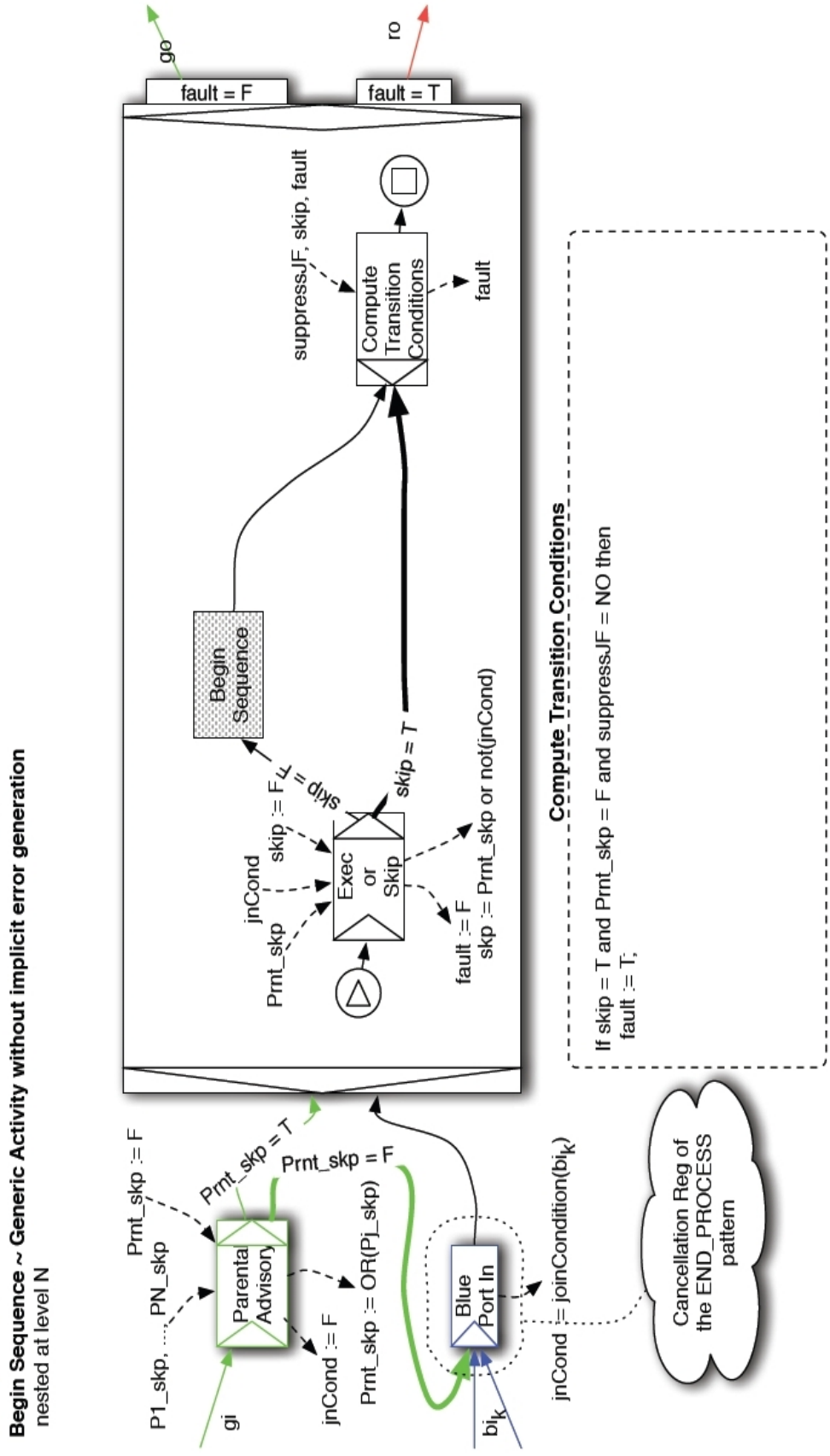

End Compensate
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
jnCond := F
Prnt_skp := OR(Pj_skp)
go (to next(compensate))
bo_k
skip
Compute Transition Conditions
bo_k
End Compensate
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

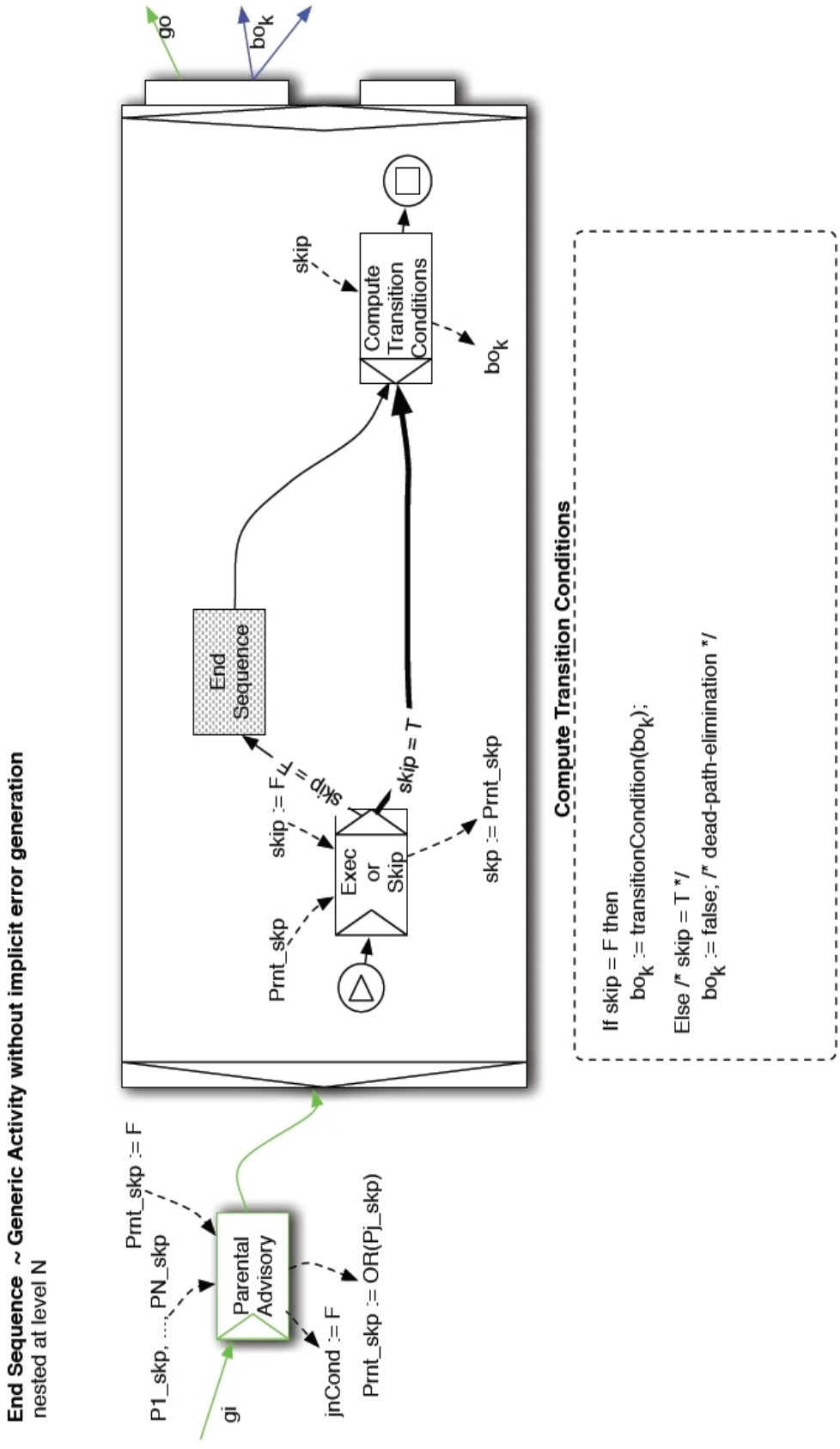

Terminate Activity ~ Generic Activity without implicit error generation
nested at level N
go
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Terminate
skip = T
skip := F, T
jnCond
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* skip = T */
bo_k := false; /* dead-path-elimination, or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pi_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
Observation: Every "terminate" activity inside the business process has to send the green token to the End Scope of the *outermost* scope, that is, the scope corresponding to the entire business process. The other important thing is the "Terminate" task should contain in its cancellation set the entire business process, except the "Begin Scope*" (the Begin Scope of the outermost scope) and the "End Scope*". That is, speaking in terms of "clouds" (i.e., patterns), every Terminate task should cancel all tokens of the "Generic Activity", "Event Handlers", and "Fault Handlers" of the outermost scope !!!


Wait Activity ~ Generic Activity without implicit error generation
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
jnCond := F
Prnt_skp := OR(Pi_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
for/until
Wait (TimeService)
jnCond
Prnt_skp
skip := F
skip := T
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
go
bo_k
ro
fault = F
fault = T
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;

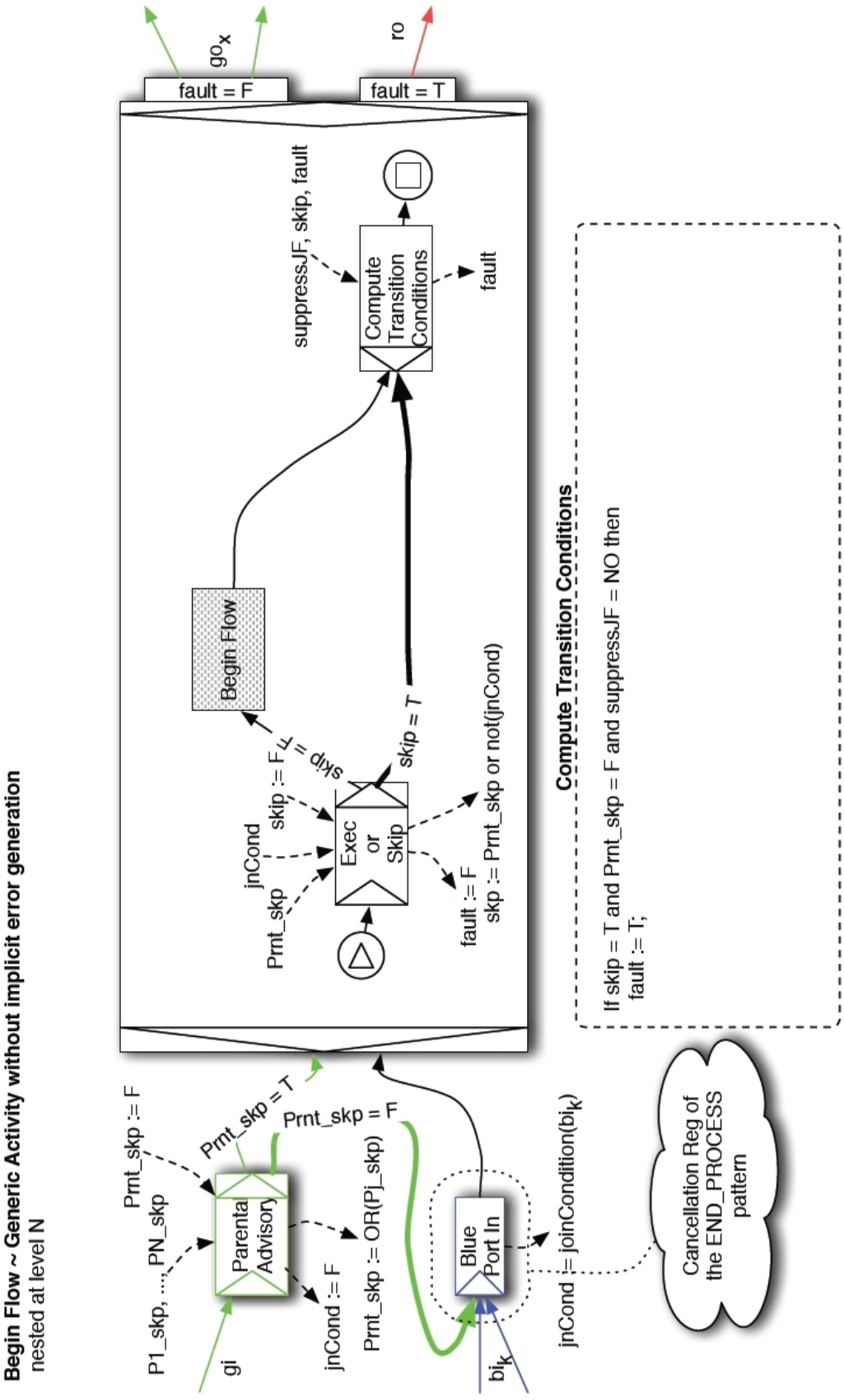

Empty Activity ~ Generic Activity without implicit error generation
nested at level N
go
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Empty
skip := F
skip := T
jnCond
Prnt_skp
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* skip = T */
bo_k := false; /* dead-path-elimination, or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern

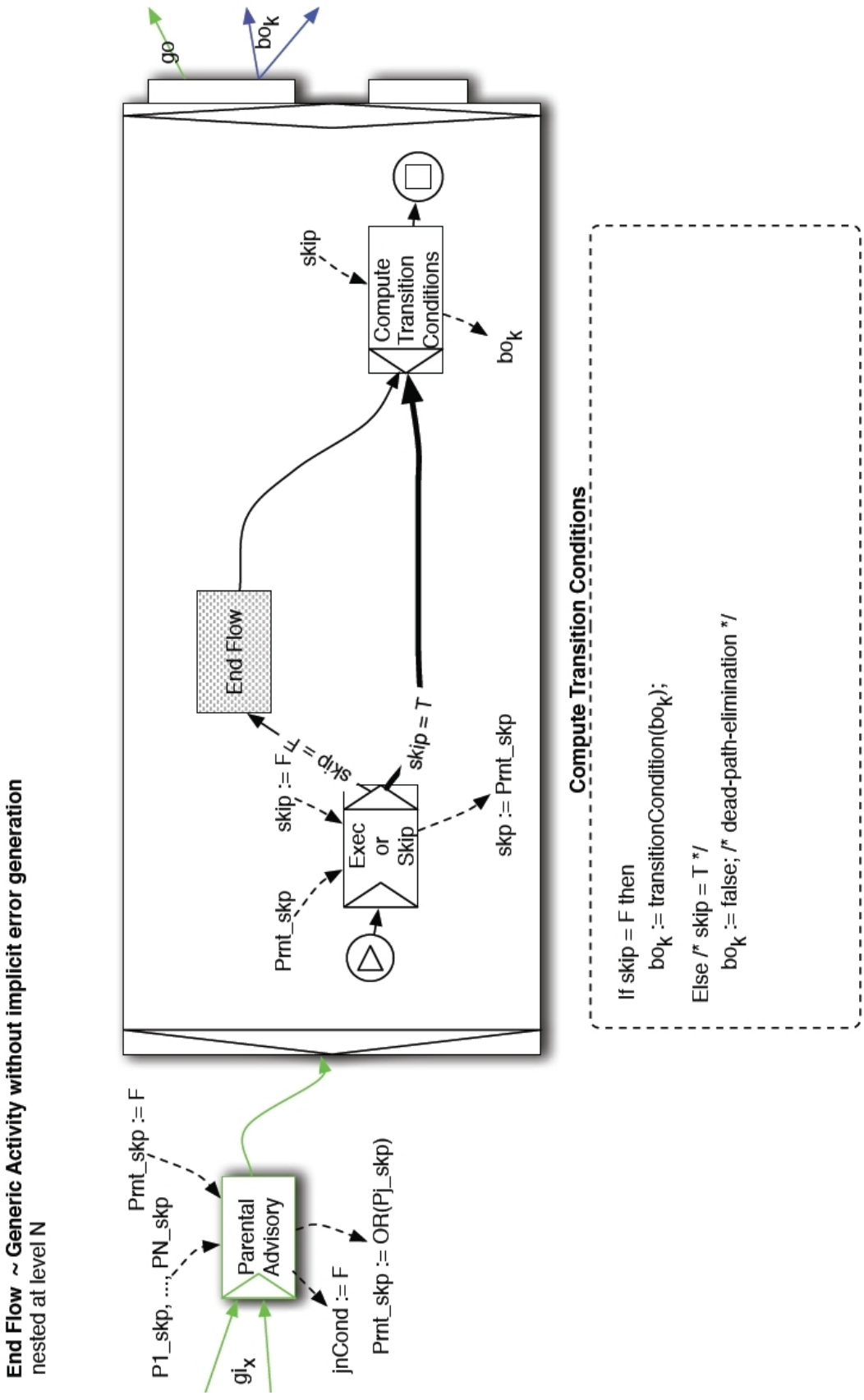
Sequence Activity = Generic Structured Activity
nested at level N
generating a fault
gi
Begin Sequence
Generic Activity
End Sequence
go
where:
Begin Sequence
=
Empty
and
End Sequence
=
Empty (without blue Is and red Os)
Observation: Begin Sequence introduces a new nesting level.
Hence, all activities inside the sequence should take into account the "skip" computed by Begin Sequence (in addition to the "skip"s for the previous nesting levels).


Begin Sequence ~ Generic Activity without implicit error generation
nested at level N
go
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin Sequence
skip := F
skip := T
skip = T
jnCond
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pi_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bik
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern

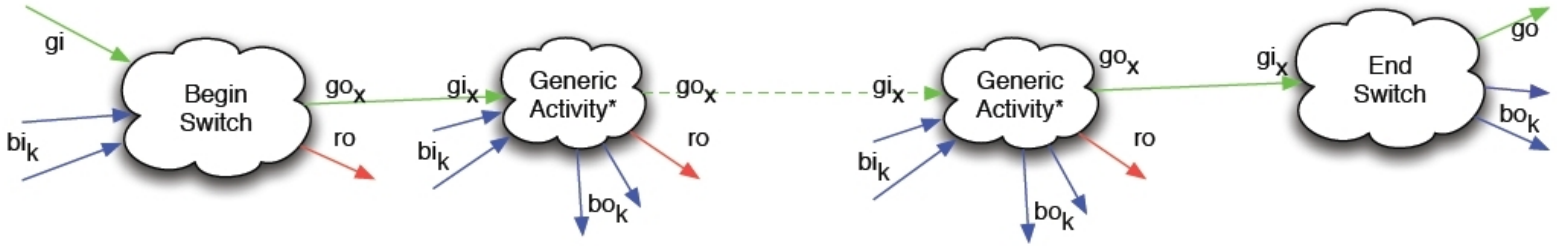

End Sequence ~ Generic Activity without implicit error generation
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
jnCond := F
Prnt_skp := OR(Pi_skp)
go
bo_k
skip
Compute Transition Conditions
bo_k
End Sequence
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

Flow Activity ~ Sequence with many green lines
nested at level N
generating a fault
gi
Begin Flow
bi_k
go_x
ro
gi
Generic Activity
bi_k
go
ro
bo_k
gi
Generic Activity
bi_k
go
ro
bo_k
gi_x
End Flow
go
bo_k
where:
Begin Flow
=
Empty*
and
Empty*
(without blue Is and red Os)
=
End Flow
Observation: Begin Flow introduces a new nesting level.
Hence, all activities inside the flow should take into account the "skip" computed by Begin Flow (in addition to the "skip"s for the previous nesting levels).
Observation: The * in "Empty*" is for the multiple green lines.

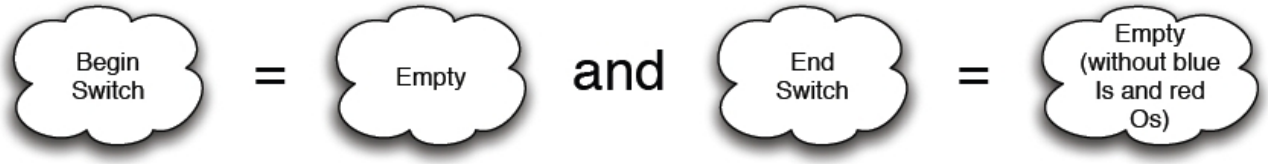

Begin Flow ~ Generic Activity without implicit error generation
nested at level N
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin Flow
jnCond
Prnt_skp
skip := F
skip = T
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parenta Advisory
jnCond := F
Prnt_skp = T
Prnt_skp = F
Prnt_skp := OR(Pj_skp)
Blue Port In
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern


End Flow ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi_x
jnCond := F
Prnt_skp := OR(Pj_skp)
go
bo_k
skip
Compute Transition Conditions
bo_k
End Flow
skip := F
skip = F
skip = T
Exec or Skip
Prnt_skp
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

Switch Activity = Generic Structured Activity
nested at level N
generating a fault



where:

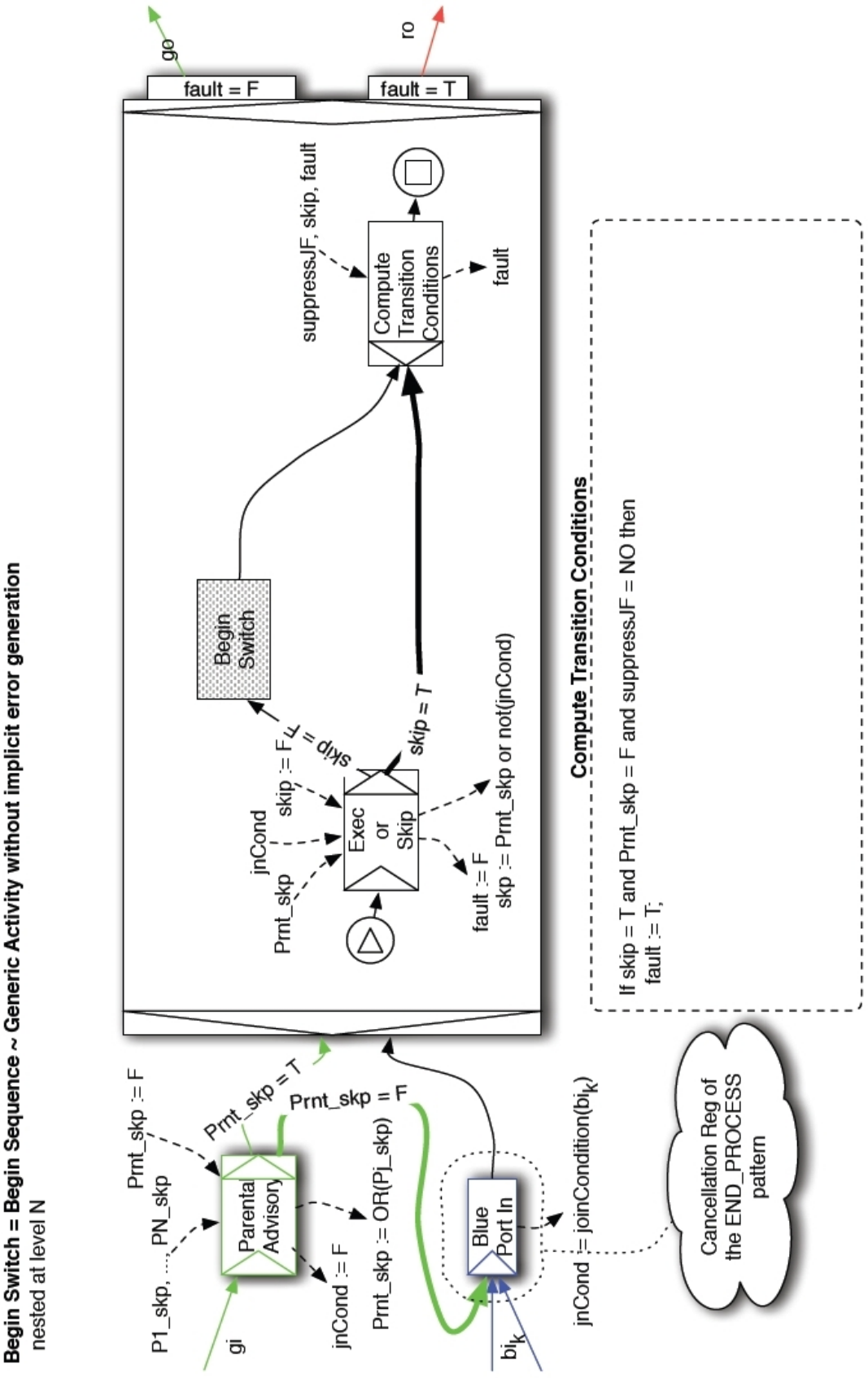


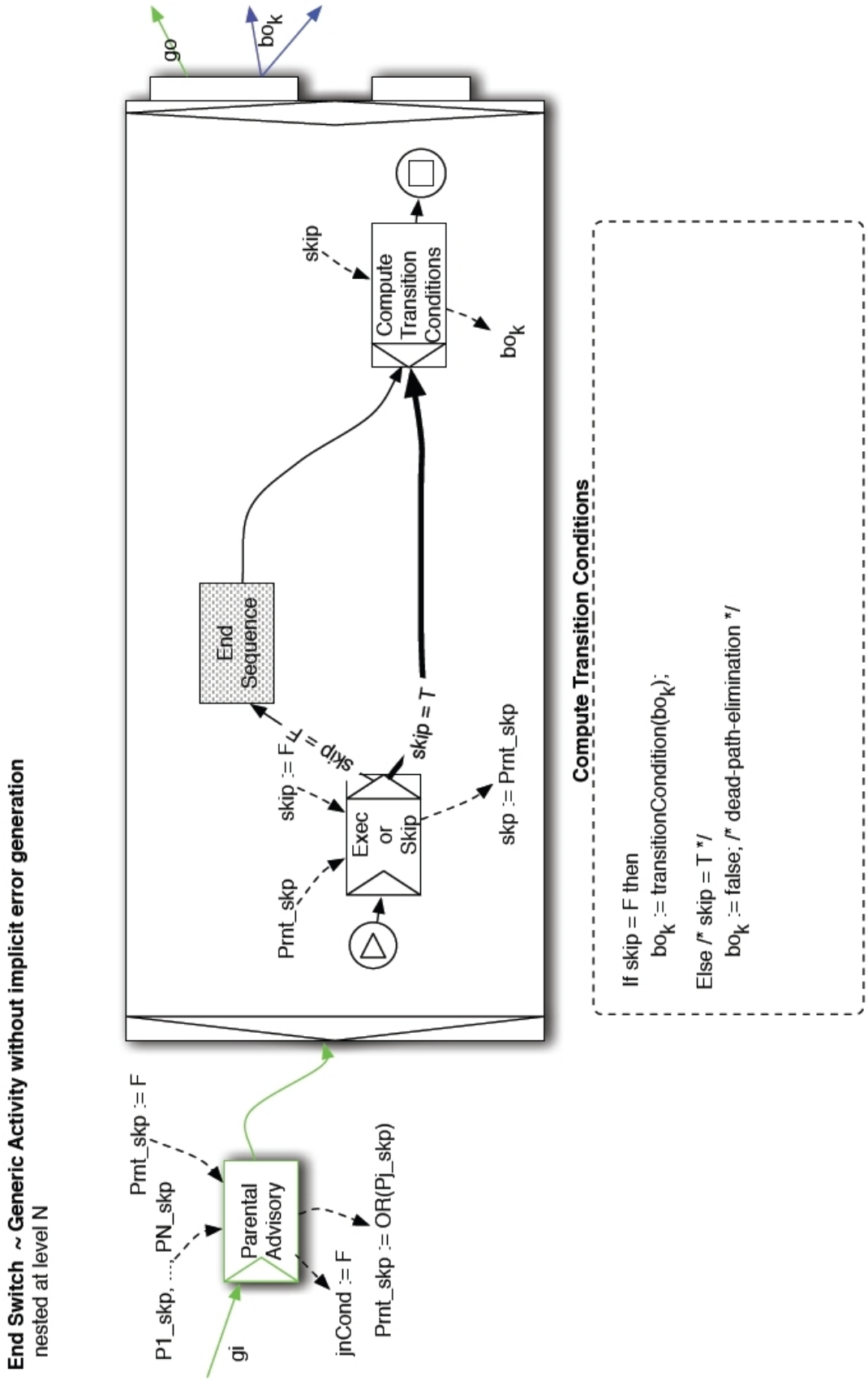
Observation: Begin Switch introduces a new nesting level. Hence, all activities inside the sequence should take into account the "skip" computed by Begin Switch (in addition to the "skip"s for the previous nesting levels).

Observation: The sequencing order is given by the structural order in the BPEL file. That is, the first case comes first and so on.

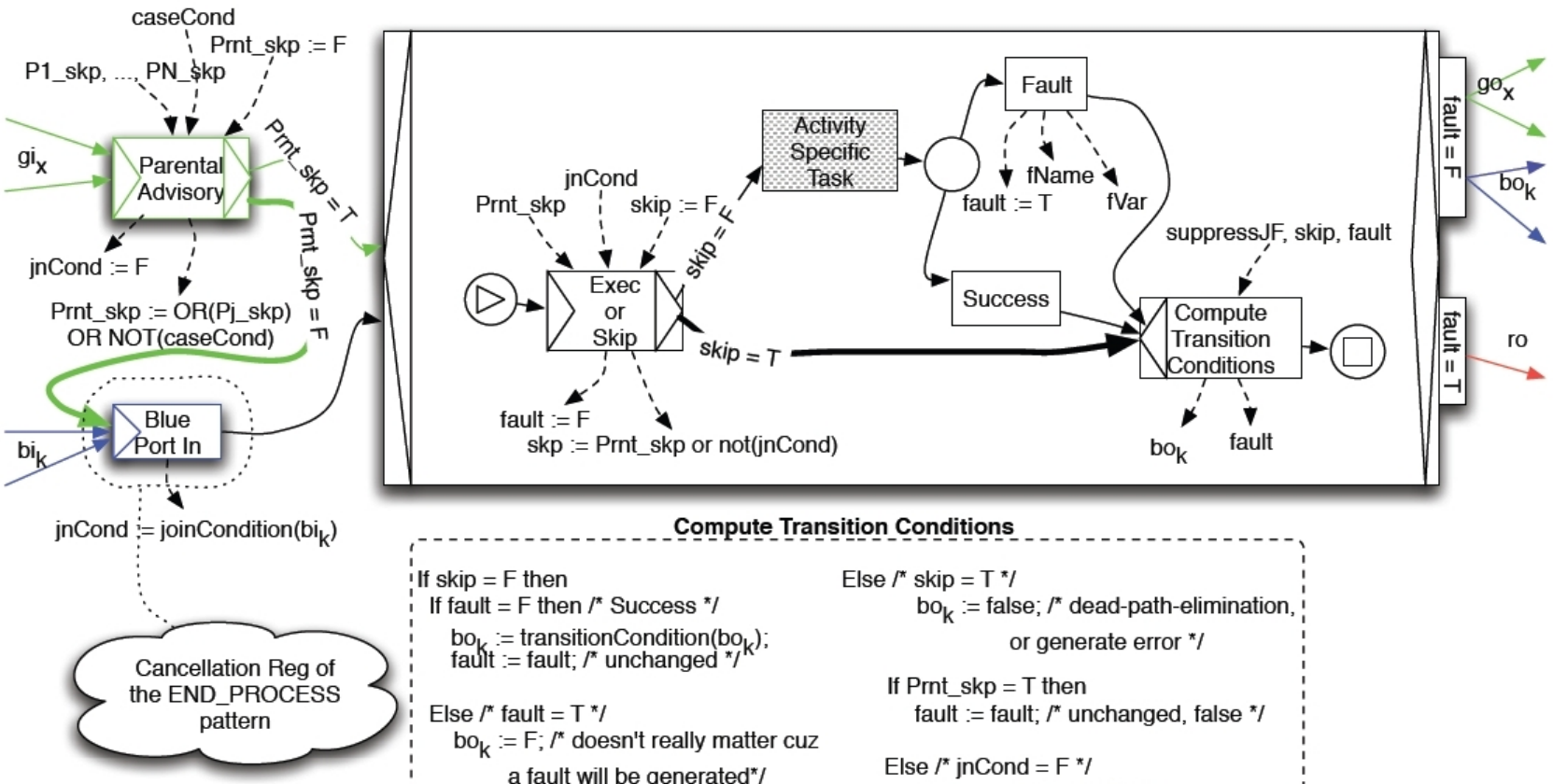
In case no Otherwise branch is given in the BPEL file, an Otherwise pattern as a Generic Activity will be added right before the End Switch.
See Generic Activity* for more details.

Observation: Instead of sequencing the cases, they could be run in parallel (as inside a Flow). See the Flow for more details.
The semantics changes a bit as in the *sequencing case*, the dead-path-elimination is done before and after the execution of the matched case, while in the *flow* case, it's done in parallel. Still, it looks like an implementation issue, as the result will be the same.


Begin Switch = Begin Sequence ~ Generic Activity without implicit error generation
nested at level N
go
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin Switch
skip := F
skip := T
skip = T
jnCond
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Prnt_skp := F
Prnt_skp = T
Prnt_skp = F
P1_skp, ..., PN_skp
Parenta Advisory
gi
jnCond := F
Prnt_skp := OR(Pj_skp)
Blue Port In
bik
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern


End Switch ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pj_skp)
go
bo_k
skip
Compute Transition Conditions
bo_k
End Sequence
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

Generic Activity*, that is, Generic Activity child of a Switch.
nested at level N
generating a fault



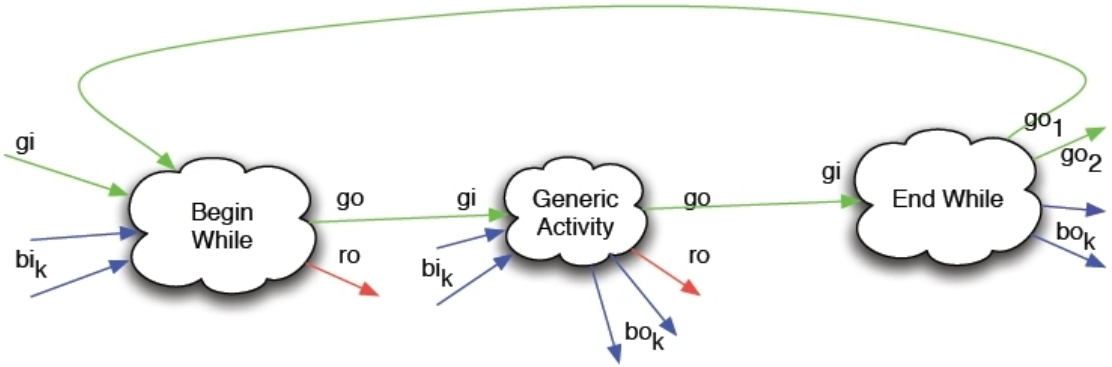
Compute Transition Conditions

```
If skip = F then
  If fault = F then /* Success */
     bo_k := transitionCondition(bo_k);
     fault := fault; /* unchanged */

Else /* fault = T */
     bo_k := F; /* doesn't really matter cuz
           a fault will be generated*/

Else /* skip = T */
        bo_k := false; /* dead-path-elimination,
                  or generate error */

   If Prnt_skp = T then
      fault := fault; /* unchanged, false */

Else /* jnCond = F */
      If suppressJF = NO then
           fault := T;
```

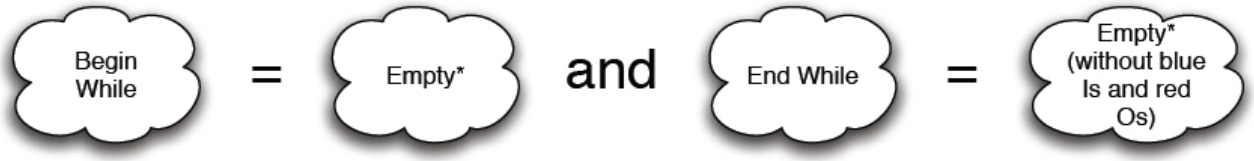
Observation: The computation of the "Prnt_skp" takes into account the condition of the case (i.e., caseCond).

Observation: In case no Otherwise branch is given in the BPEL file, an Otherwise pattern with a *true* "caseCond" should be used.

While Activity = Structured Activity ~ Sequence++
nested at level N
generating a fault



where:



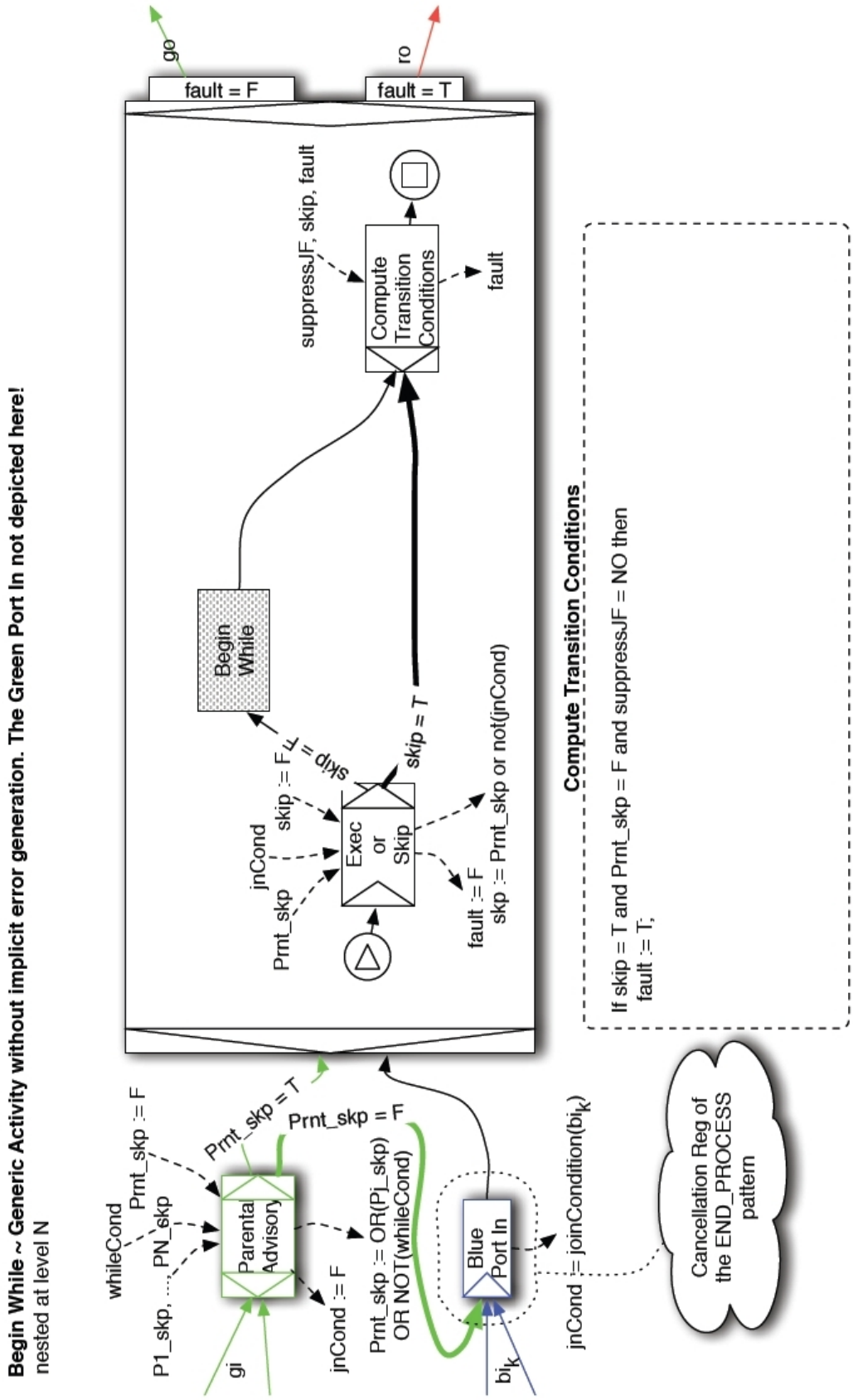
Observation: Begin While introduces a new nesting level.
Hence, all activities inside the while should take into account the "skip" computed by Begin While (in addition to the "skip"s for the previous nesting levels).

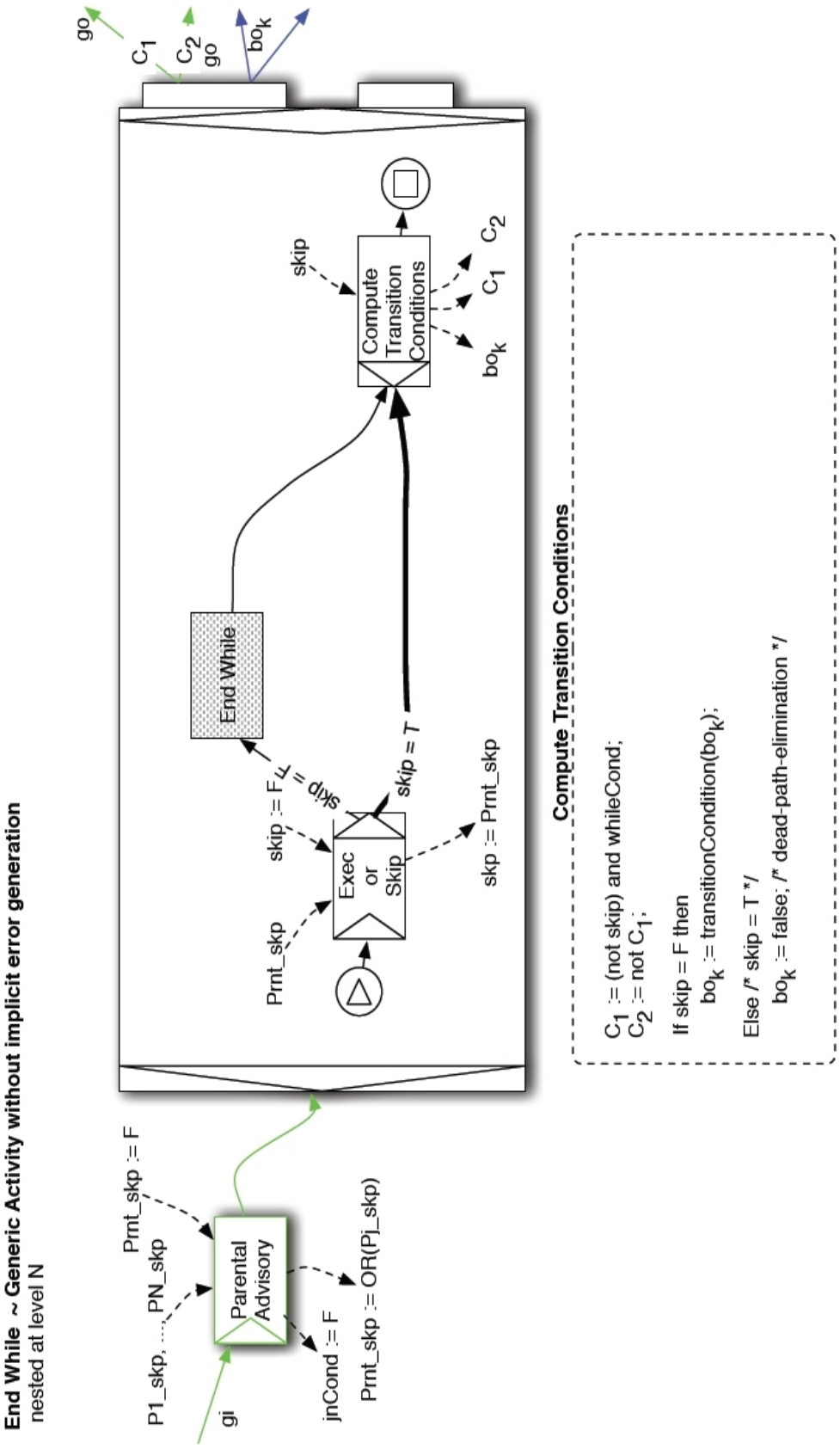
Observation: Begin While has to take into account the "while condition" -- that's why the "Empty*" pattern.

Observation: End While has to take into account the "while condition" -- that's why the "Empty*" pattern.

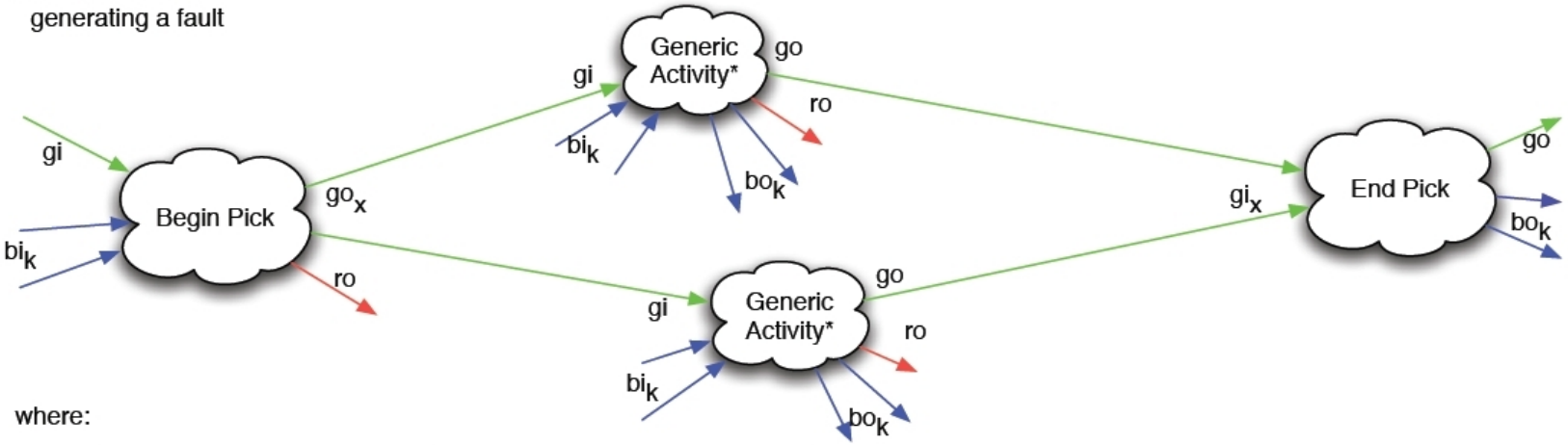
Observation: The following is incorrect !!!
Links can't cross the boundary of a while!

```
<flow>

<while cond>
<A source linkName="X">
</while>

<B target linkName="X">
</flow>
```

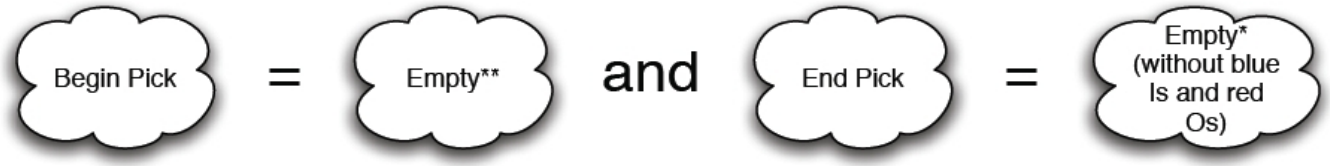

Begin While ~ Generic Activity without implicit error generation. The Green Port In not depicted here!
nested at level N
go
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin While
skip = T
skip := F
jnCond
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
whileCond
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
Prnt_skp = T
Prnt_skp = F
jnCond := F
Prnt_skp := OR(Pj_skp)
OR NOT(whileCond)
Blue Port In
bik
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern


End While ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi
jnCond := F
Prnt_skp := OR(Pj_skp)
go
C1
C2
go
bok
skip
Compute Transition Conditions
C2
C1
bok
End While
skip := F
T = skip
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
C1 := (not skip) and whileCond;
C2 := not C1;
If skip = F then
bok := transitionCondition(bok);
Else /* skip = T */
bok := false; /* dead-path-elimination */
Observation: We check the whileCond here as well instead of only in the Begin While,
so that, in the case of a "false", we don't skip the entire while.

Pick Activity ~ Flow except that only one activity will be exec and the others will be skipped
nested at level N
generating a fault



where:

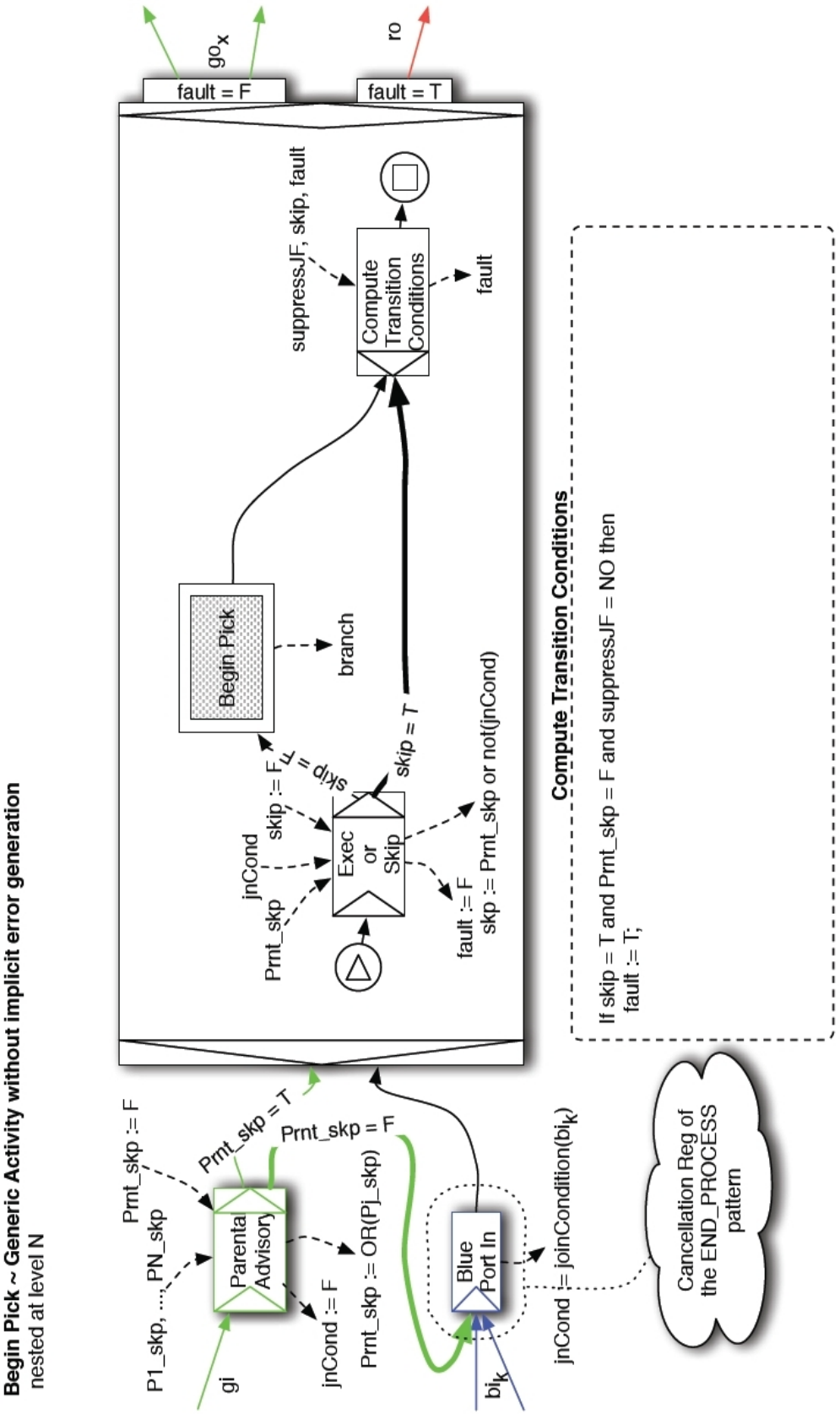


Observation: Begin Flow introduces a new nesting level.
Hence, all activities inside the flow should take into account the "skip" computed by Begin Flow (in addition to the "skip"s for the previous nesting levels).

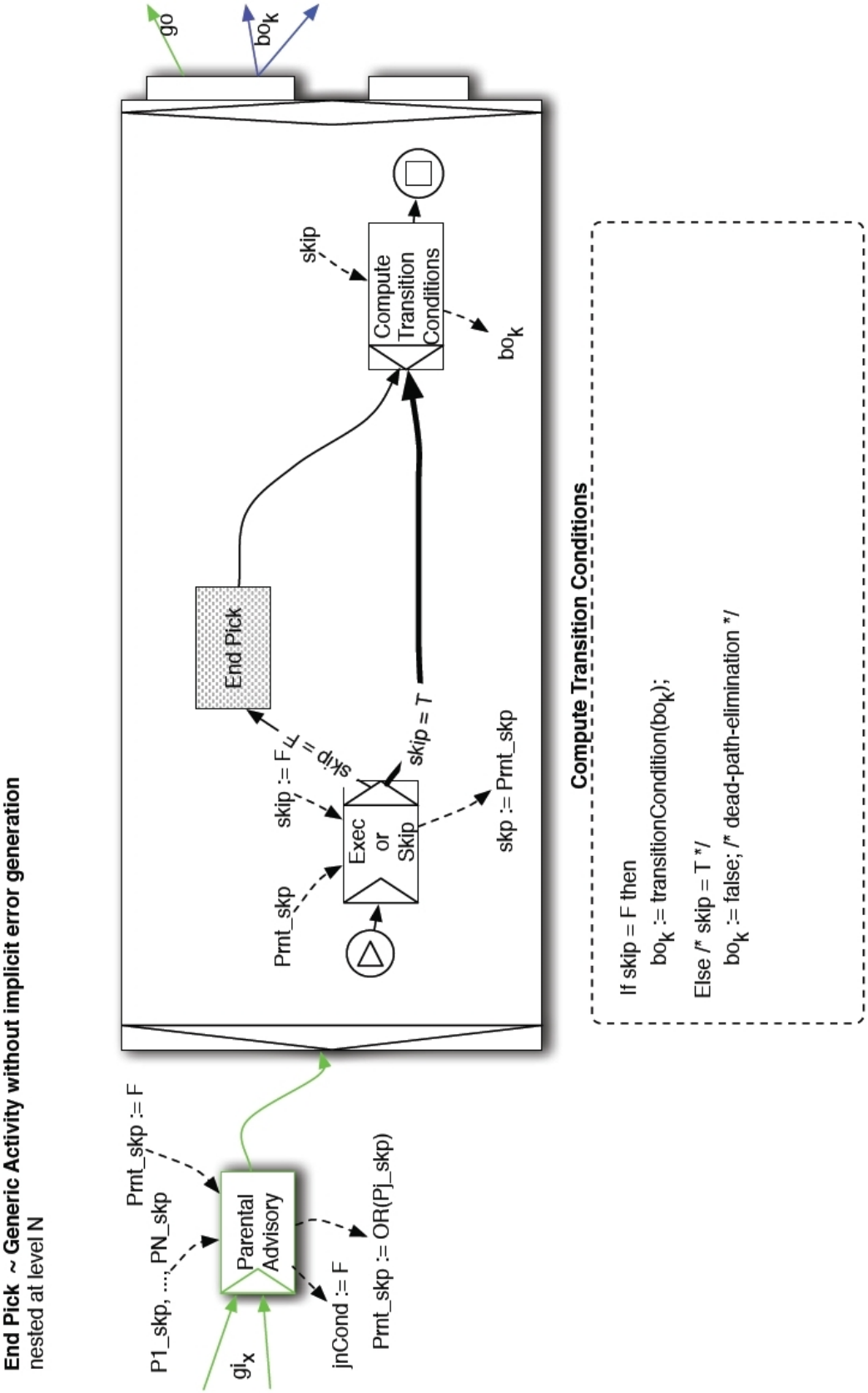
Observation: The first * in "Empty**" is for the multiple green lines.

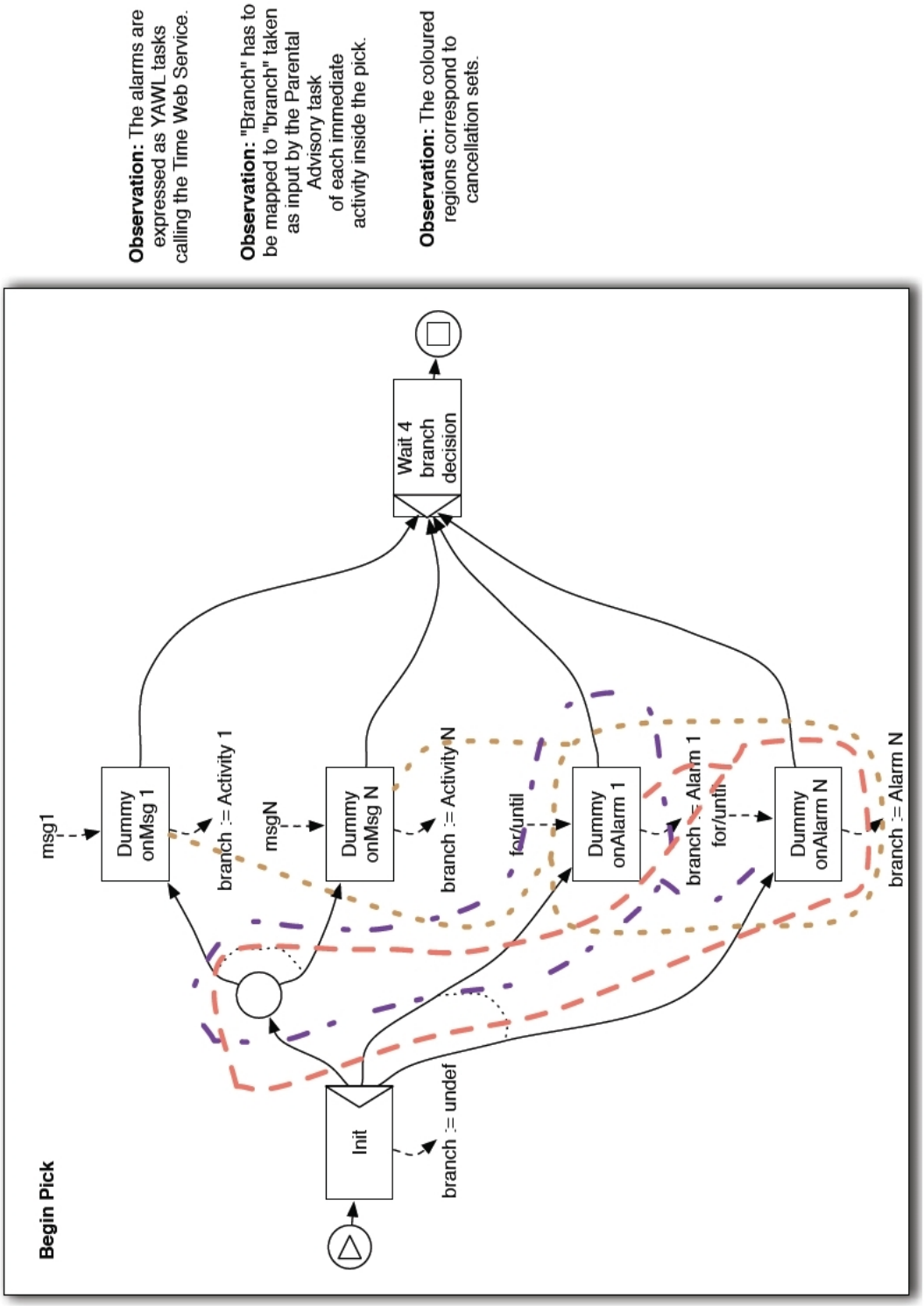
Observation: The second * in "Empty**" is because the Activity Specific Task (i.e., Begin Pick) is a *composite task*.

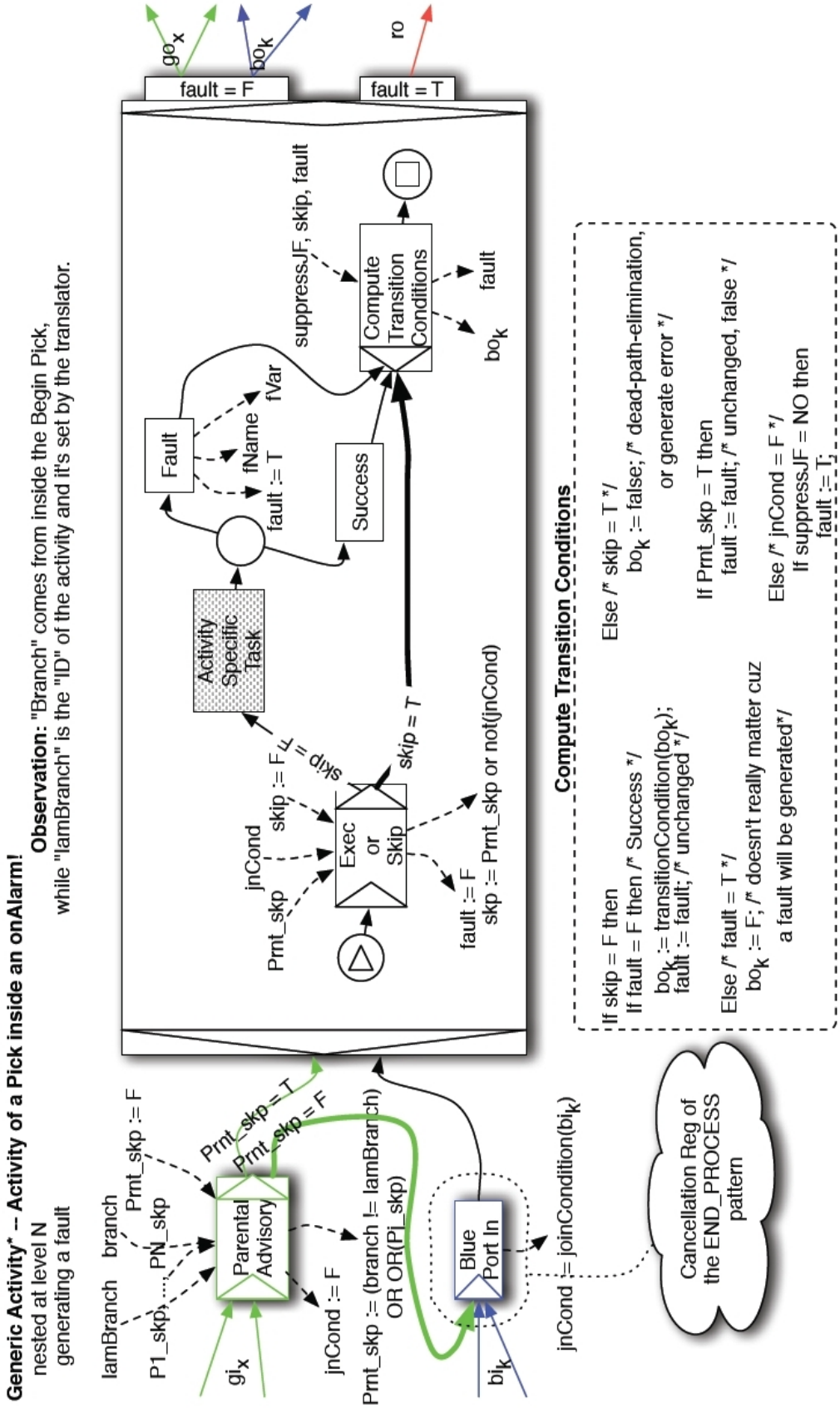
Observation: The Parental Advisory task of each "Generic Activity*" has to check whether the respective activity was selected for execution.

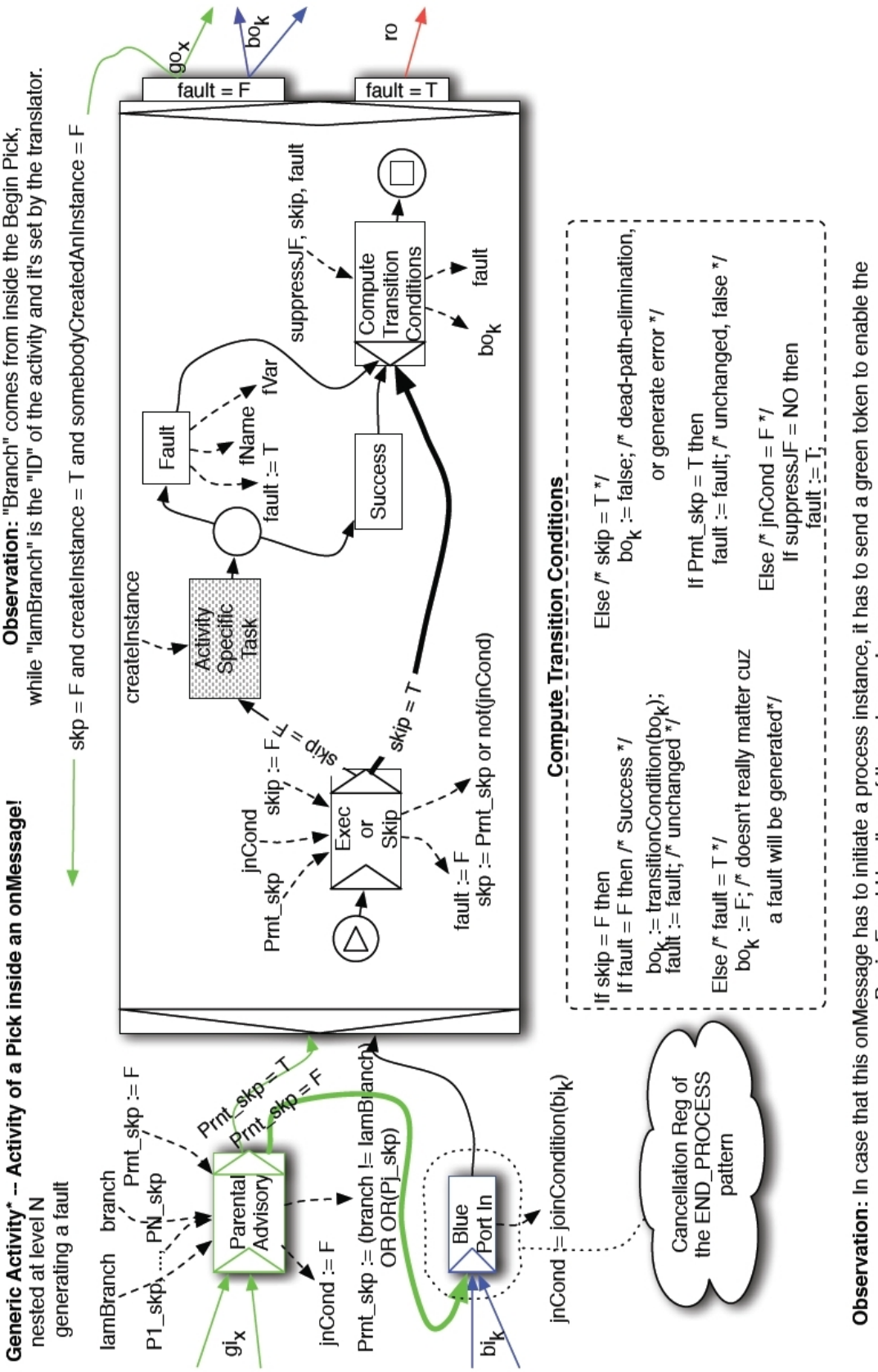


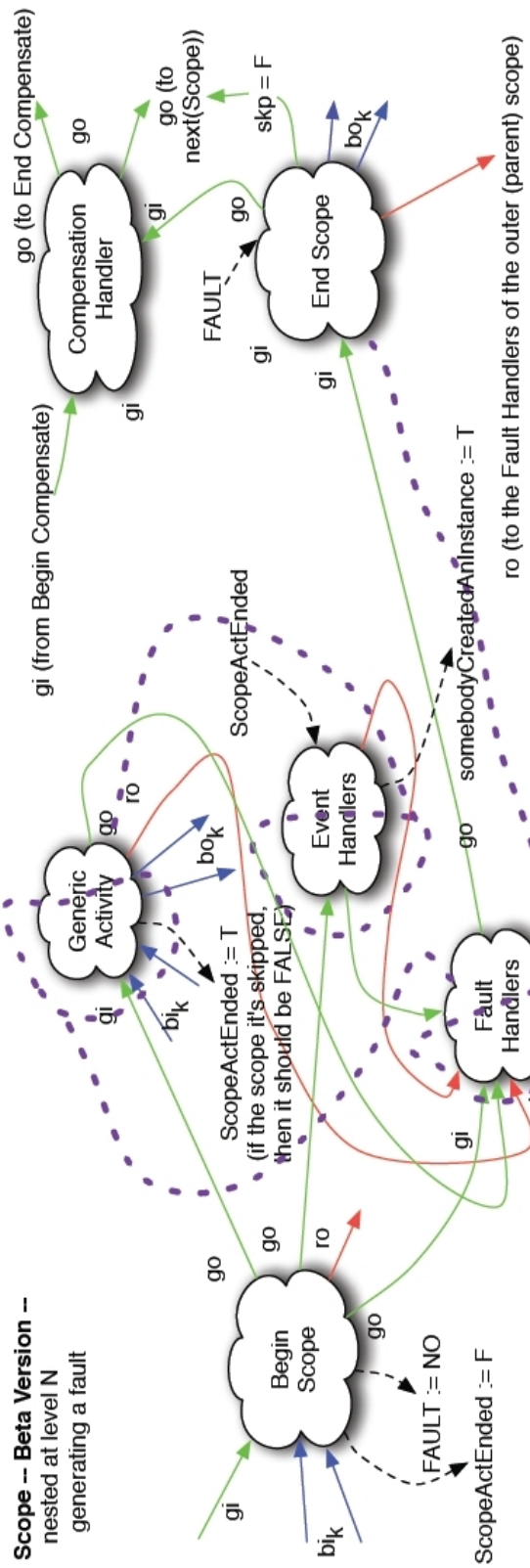
Observation: The "branch" output of Begin Pick gives the ID of the chosen activity.
Then, each (immediate) activity inside the pick has to test whether it was the chosen one.

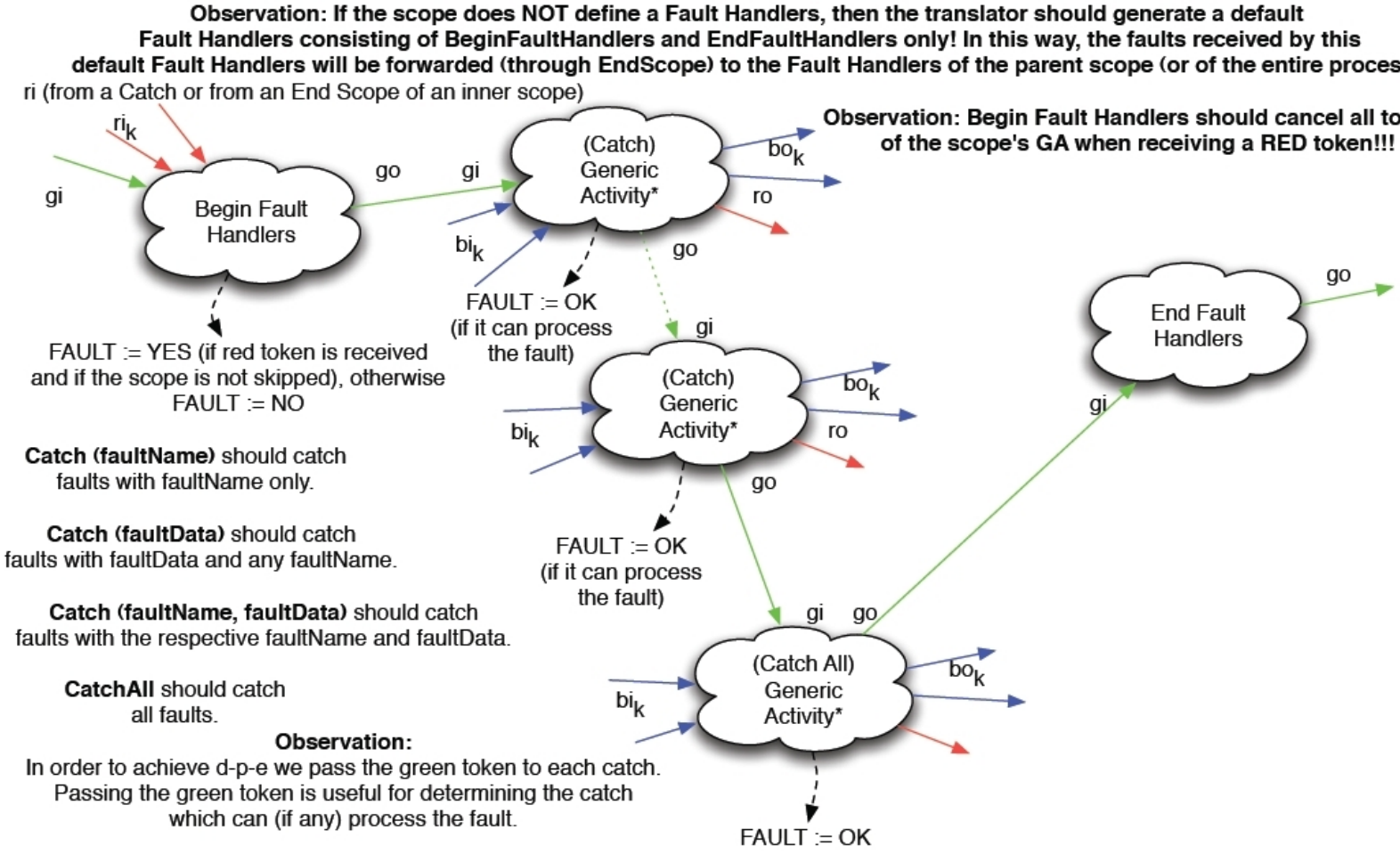

End Pick ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gi_x
jnCond := F
Prnt_skp := OR(Pj_skp)
go
bo_k
skip
Compute Transition Conditions
bo_k
End Pick
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
If skip = F then
bo_k := transitionCondition(bo_k);
Else /* skip = T */
bo_k := false; /* dead-path-elimination */

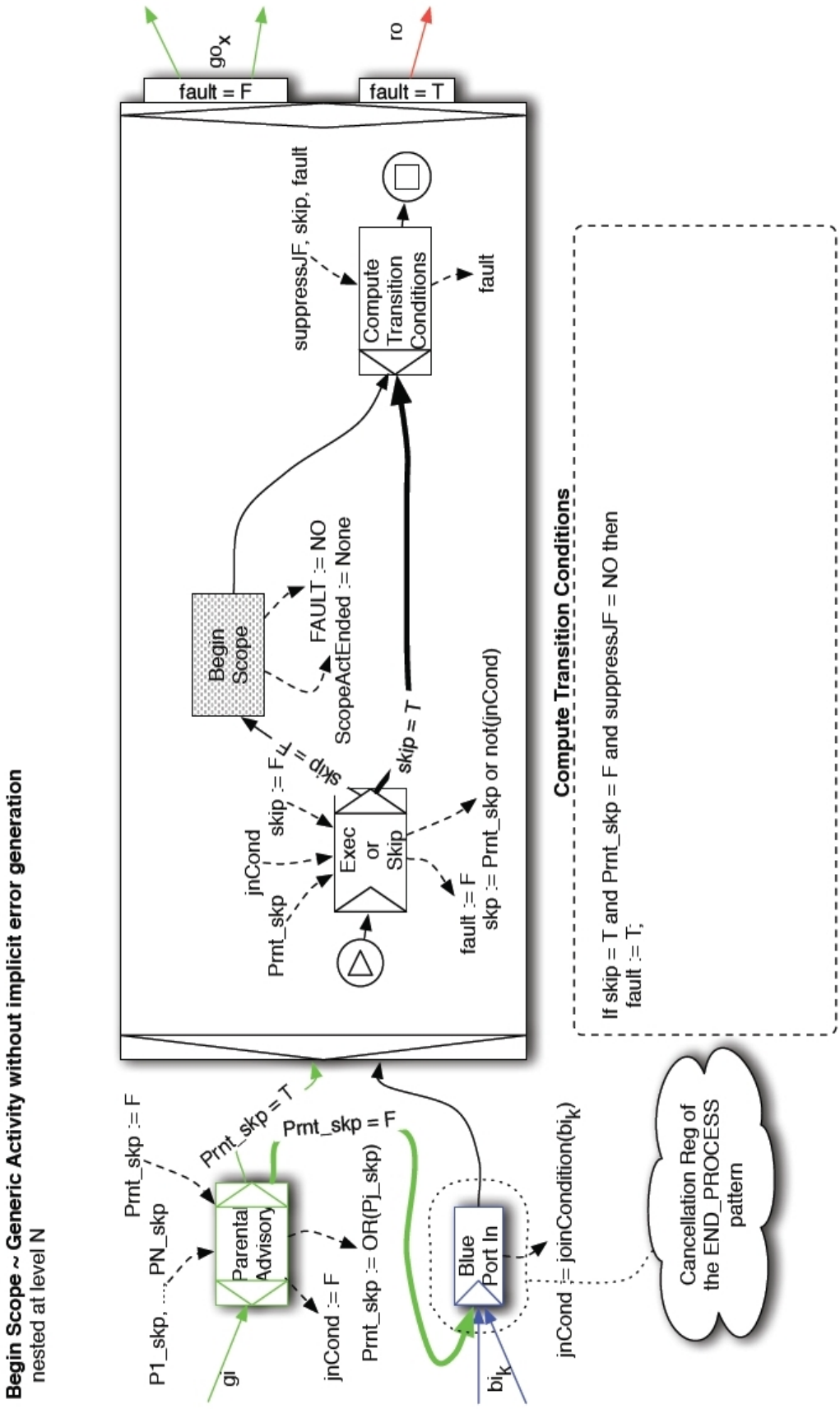

Observation: The alarms are expressed as YAWL tasks calling the Time Web Service.
Observation: "Branch" has to be mapped to "branch" taken as input by the Parental Advisory task of each immediate activity inside the pick.
Observation: The coloured regions correspond to cancellation sets.
Begin Pick
Init
branch := undef
msg1
Dummy onMsg 1
branch := Activity 1
msgN
Dummy onMsg N
branch := Activity N
for/until
Dummy onAlarm 1
branch := Alarm 1
for/until
Dummy onAlarm N
branch := Alarm N
Wait 4 branch decision

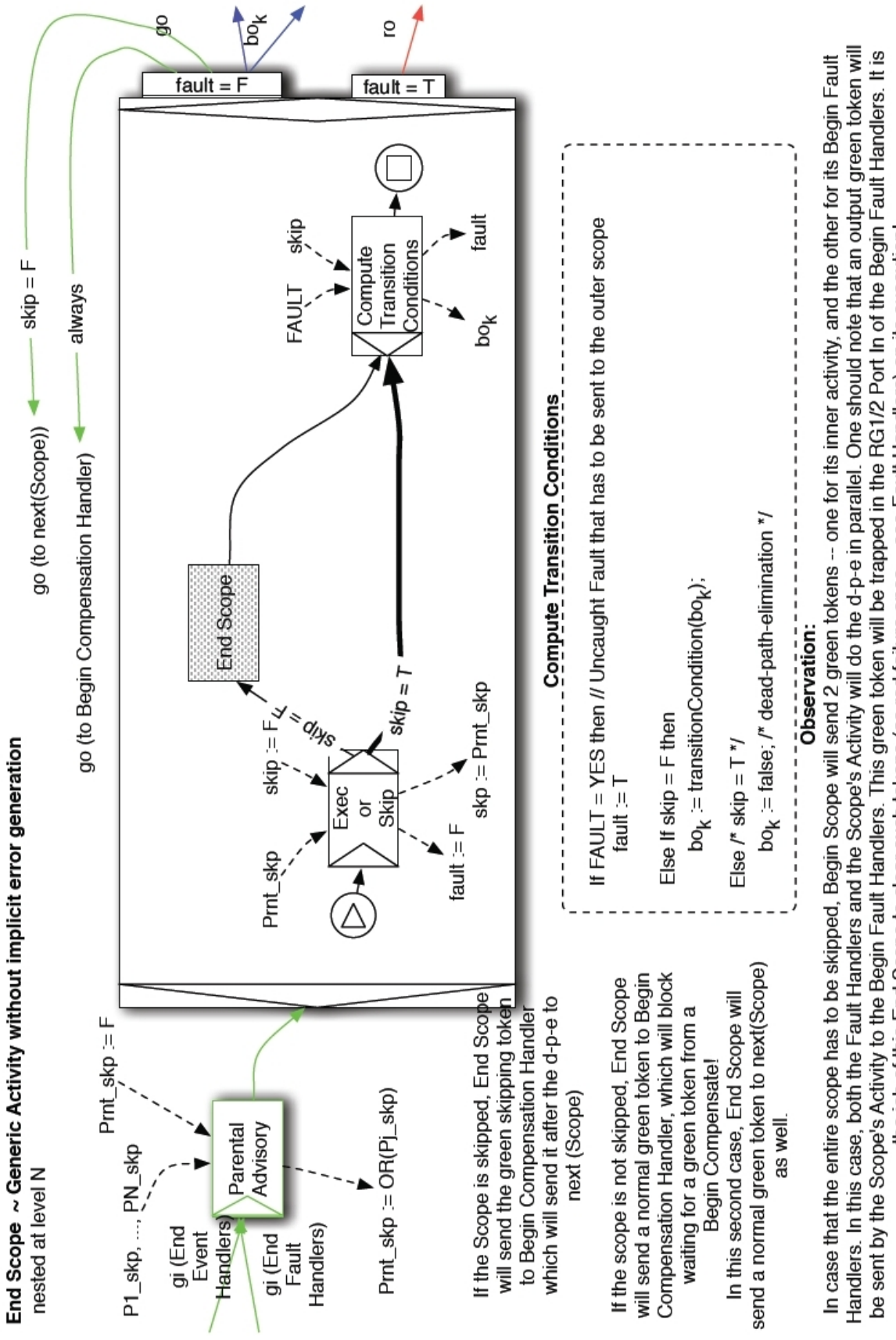

Generic Activity* -- Activity of a Pick inside an onAlarm!
nested at level N
generating a fault
Observation: "lamBranch" comes from inside the Begin Pick,
while "lamBranch" is the "ID" of the activity and it's set by the translator.
lamBranch
branch
P1_skp, ..., PN_skp
Prnt_skp := F
Parental Advisory
gi_x
jnCond := F
Prnt_skp := (branch != lamBranch) OR OR(Pi_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
go_x
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
Fault
fName
fVar
fault := T
Success
Activity Specific Task
jnCond
Prnt_skp
skip := F
skip = T
Exec or Skip
skip = T
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;


Generic Activity* -- Activity of a Pick inside an onMessage!
nested at level N
generating a fault
Observation: "Branch" comes from inside the Begin Pick,
while "lamBranch" is the "ID" of the activity and it's set by the translator.
skp = F and createInstance = T and somebodyCreatedAnInstance = F
lamBranch branch
P1_skp, ..., PN_skp
Prnt_skp := F
Parental Advisory
jnCond := F
Prnt_skp := (branch != lamBranch)
OR OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
Blue Port In
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
createInstance
Activity Specific Task
Fault
fName
fVar
fault := T
Success
suppressJF, skip, fault
Compute Transition Conditions
fault
bo_k
fault = F
fault = T
go_x
bo_k
ro
jnCond
Prnt_skp
skip := F
skip = T
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
skip = T
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Observation: In case that this onMessage has to initiate a process instance, it has to send a green token to enable the
Begin Event Handlers of the outermost scope.


Scope -- Beta Version --
nested at level N
generating a fault
gi
Begin Scope
go
go
FAULT := NO
ScopeActEnded := F
gi (from Begin Compensate)
go (to End Compensate)
Generic Activity
ro
ScopeActEnded := T
(if the scope it's skipped, then it should be FALSE)
Event Handlers
ScopeActEnded
Compensation Handler
go (to next(Scope))
FAULT
skp = F
End Scope
somebodyCreatedAnInstance := T
ro (to the Fault Handlers of the outer (parent) scope)
Fault Handlers
FAULT
Observation: Begin Fault Handlers should cancel all tokens of the scope's GA (except tokens of CompensationHandlers) when receiving a RED token!!!
somebodyCreatedAnInstance becomes true when a receive for e.g., sends a green token to the Event Handlers. It should be defined for the outermost Event Handlers only!
where:
Begin Scope
Empty*
End Scope
Empty* (without blue Is and red Os)
Observation: The * in "Empty*" is for the multiple green lines.
Observation: Begin Scope introduces a new nesting level.
Hence, the activity inside the scope should take into account the "skip" computed by Begin Scope (in addition to the "skip"s for the previous nesting levels).
Observation: Begin Scope enables (green line) both the Fault Handlers and the Event Handlers.
The Event Handler receives a green token iff the scope is not skipped -- so as to not cycle and skip forever its activities. A link cannot cross it's boundary so it's no need 4 dead-path-elim. inside it !!!
On the other hand, the Fault Handler always gets a green token so as to achieve the d-p-e when the scope is skipped !!!
Observation: GA and FH are in a *sequence* as, "in the worst case" the FH has to be skipped if the GA completes its exec successfully! If the entire scope gets skipped, then GA and FH will be skipped in parallel. In this case, the output green token from GA will be trapped in the RG1/2 Port In of the BFH yet cleared later by the End Scope!
Fault Handlers


Observation: If the scope does NOT define a Fault Handlers, then the translator should generate a default Fault Handlers consisting of BeginFaultHandlers and EndFaultHandlers only! In this way, the faults received by this default Fault Handlers will be forwarded (through EndScope) to the Fault Handlers of the parent scope (or of the entire process).
ri (from a Catch or from an End Scope of an inner scope)
rik
gi
Begin Fault Handlers
go
gi
(Catch) Generic Activity*
bok
ro
bik
go
Observation: Begin Fault Handlers should cancel all tokens of the scope's GA when receiving a RED token!!!
FAULT := YES (if red token is received and if the scope is not skipped), otherwise FAULT := NO
FAULT := OK (if it can process the fault)
gi
(Catch) Generic Activity*
bok
ro
bik
go
FAULT := OK (if it can process the fault)
End Fault Handlers
go
gi
Catch (faultName) should catch faults with faultName only.
Catch (faultData) should catch faults with faultData and any faultName.
Catch (faultName, faultData) should catch faults with the respective faultName and faultData.
CatchAll should catch all faults.
gi
go
(Catch All) Generic Activity*
bok
bik
FAULT := OK
Observation:
In order to achieve d-p-e we pass the green token to each catch. Passing the green token is useful for determining the catch which can (if any) process the fault.
If a Catch cannot process a fault, it should skip its inner activity (and hence d-p-e is ok) and it should pass the green token to the next catch in line.
The last catch -- if it's not a Catch All -- should send the green token to the End Fault Handlers.

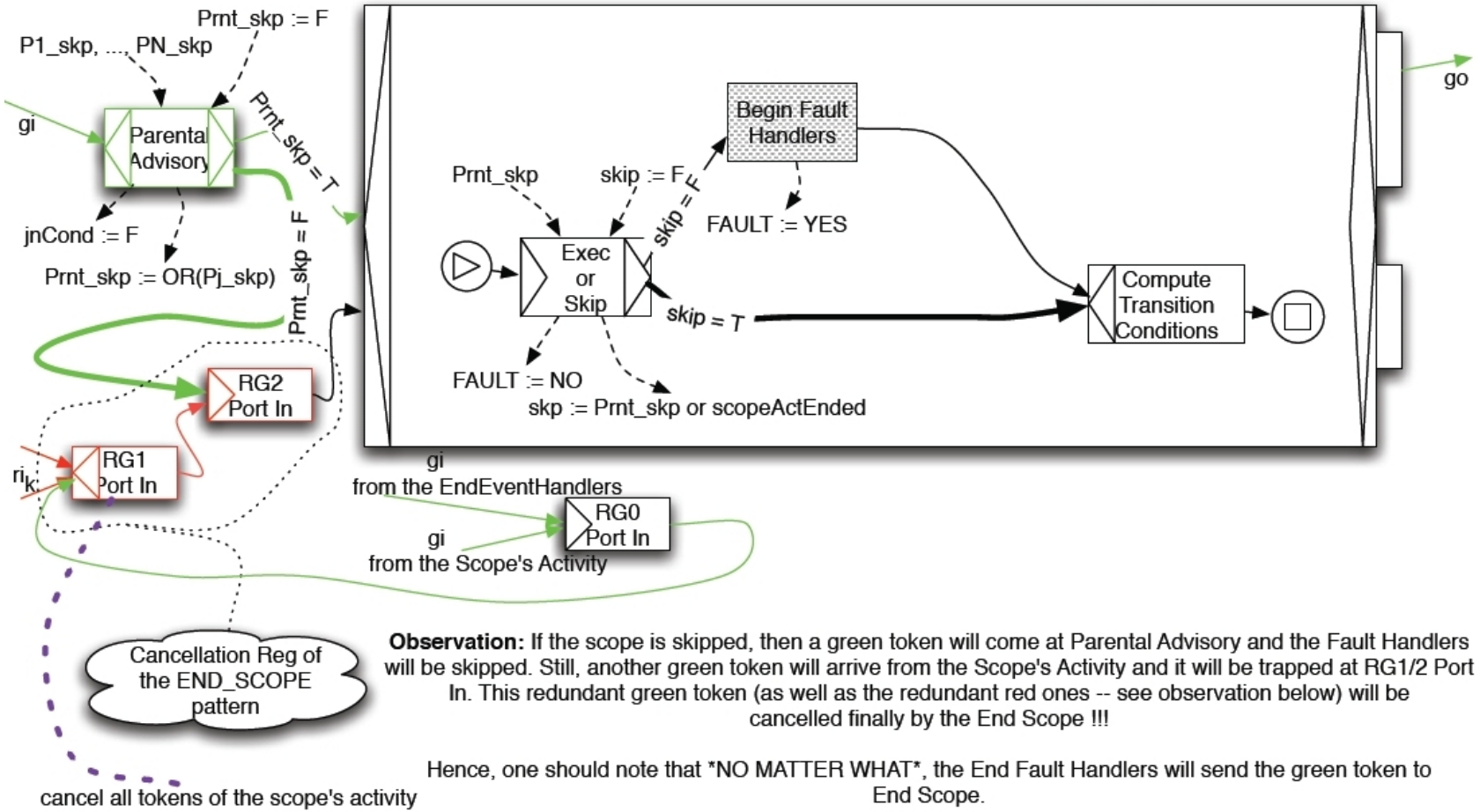

Begin Scope ~ Generic Activity without implicit error generation
nested at level N
fault = F
fault = T
suppressJF, skip, fault
Compute Transition Conditions
fault
Begin Scope
FAULT := NO
ScopeActEnded := None
skip = T
skip = F
jnCond
Prnt_skp
skip := F
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = T and Prnt_skp = F and suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
Parental Advisory
gI
jnCond := F
Prnt_skp = T
Prnt_skp = F
Prnt_skp := OR(Pj_skp)
Blue Port In
jnCond := joinCondition(bik)
Cancellation Reg of the END_PROCESS pattern


End Scope ~ Generic Activity without implicit error generation
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi (End Event Handlers)
gi (End Fault Handlers)
Parental Advisory
Prnt_skp := OR(Pi_skp)
go (to next(Scope))
skip = F
go (to Begin Compensation Handler)
always
go
bok
ro
fault = F
fault = T
End Scope
Prnt_skp
skip := F
skip = T
Exec or Skip
fault := F
skp := Prnt_skp
skip = T
FAULT
skip
Compute Transition Conditions
fault
bok
If the Scope is skipped, End Scope will send the green skipping token to Begin Compensation Handler which will send it after the d-p-e to next (Scope)
If the scope is not skipped, End Scope will send a normal green token to Begin Compensation Handler, which will block waiting for a green token from a Begin Compensate!
In this second case, End Scope will send a normal green token to next(Scope) as well.
Compute Transition Conditions
If FAULT = YES then // Uncaught Fault that has to be sent to the outer scope
fault := T
Else If skip = F then
bok := transitionCondition(bok);
Else /* skip = T */
bok := false; /* dead-path-elimination */
Observation:
In case that the entire scope has to be skipped, Begin Scope will send 2 green tokens -- one for its inner activity, and the other for its Begin Fault Handlers. In this case, both the Fault Handlers and the Scope's Activity will do the d-p-e in parallel. One should note that an output green token will be sent by the Scope's Activity to the Begin Fault Handlers. This green token will be trapped in the RG1/2 Port In of the Begin Fault Handlers. It is the job of this End Scope to clear such tokens (as red failure ones -- see Fault Handlers) on its execution!

Begin Fault Handlers ~ Generic Activity without implicit error generation
nested at level N

Observation: Begin Fault Handlers should cancel all tokens of the scope's GA (except inputs of the CompensationHandlers) when receiving a RED token!!!

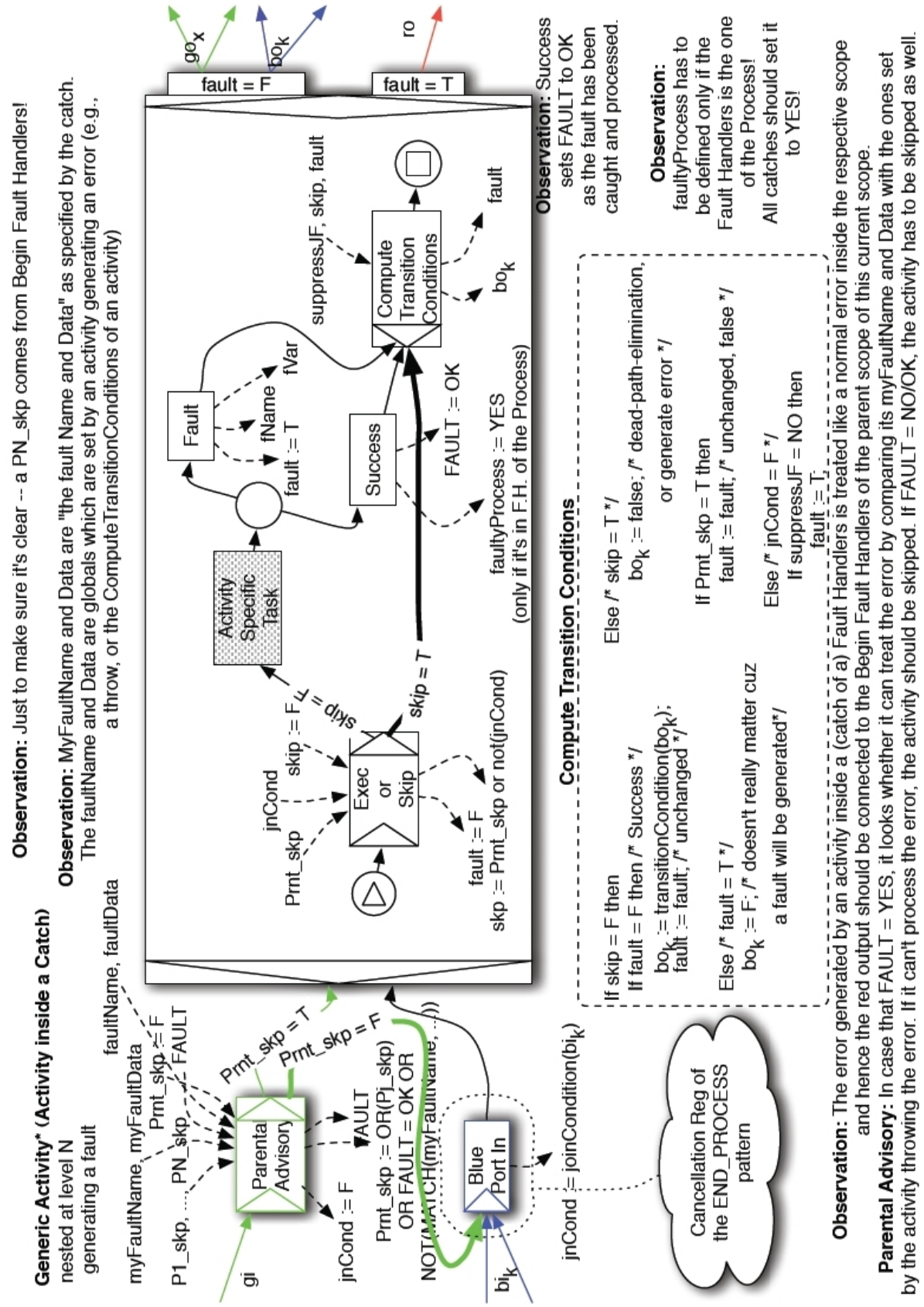
(To be more exact, RG1 Port In could do the dirty job)



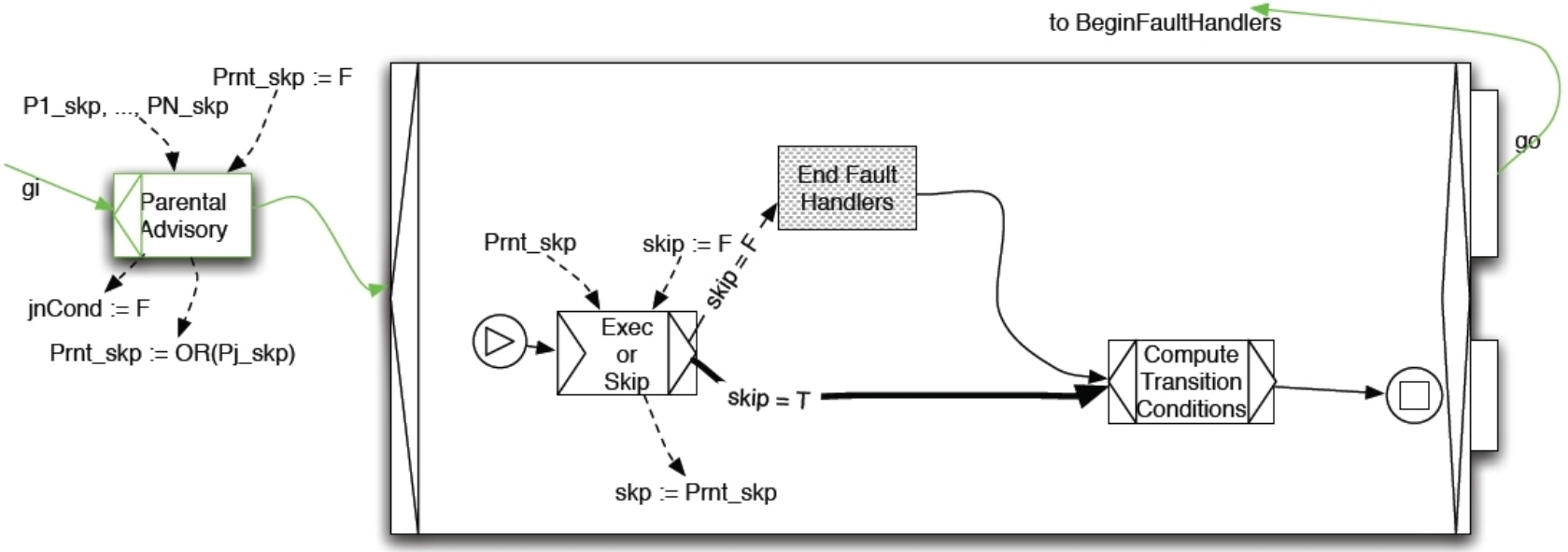
Observation: If the scope is skipped, then a green token will come at Parental Advisory and the Fault Handlers will be skipped. Still, another green token will arrive from the Scope's Activity and it will be trapped at RG1/2 Port In. This redundant green token (as well as the redundant red ones -- see observation below) will be cancelled finally by the End Scope !!!

Hence, one should note that *NO MATTER WHAT*, the End Fault Handlers will send the green token to End Scope.

Observation: In case the Scope's Activity successfully completes (i.e., scopeActEnded = T), we have to emloy the d-p-e inside the Fault Handlers!
Observation: Cancel the red tokens from the RG1 Port In and from the RG2 Port In, as a red token may arrive while another fault is being treated.


Observation: Just to make sure it's clear -- a PN_skp comes from Begin Fault Handlers!
Generic Activity* (Activity inside a Catch)
nested at level N
generating a fault
Observation: MyFaultName and Data are "the fault Name and Data" as specified by the catch.
The faultName and Data are globals which are set by an activity generating an error (e.g.,
a throw, or the ComputeTransitionConditions of an activity)
faultName, faultData
myFaultName, myFaultData
P1_skp, ..., PN_skp
Prnt_skp := F
FAULT
gi
Parental Advisory
Prnt_skp = T
Prnt_skp = F
jnCond := F
FAULT
Prnt_skp := OR(Pj_skp)
OR FAULT = OK OR
NOT(MATCH(myFaultName, ...))
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
fault = F
fault = T
go_x
bo_k
ro
Fault
fName
fVar
fault := T
Activity Specific Task
jnCond
Prnt_skp
skip := F
skip = T
Exec or Skip
fault := F
skp := Prnt_skp or not(jnCond)
Success
FAULT := OK
faultyProcess := YES
(only if it's in F.H. of the Process)
suppressJF, skip, fault
Compute Transition Conditions
bo_k
fault
Observation: Success sets FAULT to OK as the fault has been caught and processed.
Compute_Transition_Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Observation:
faultyProcess has to be defined only if the Fault Handlers is the one of the Process!
All catches should set it to YES!
Observation: The error generated by an activity inside a (catch of a) Fault Handlers is treated like a normal error inside the respective scope and hence the red output should be connected to the Begin Fault Handlers of the parent scope of this current scope.
Parental Advisory: In case that FAULT = YES, it looks whether it can treat the error by comparing its myFaultName and Data with the ones set by the activity throwing the error. If it can't process the error, the activity should be skipped. If FAULT = NO/OK, the activity has to be skipped as well.

End Fault Handlers ~ Generic Activity without implicit error generation
nested at level N



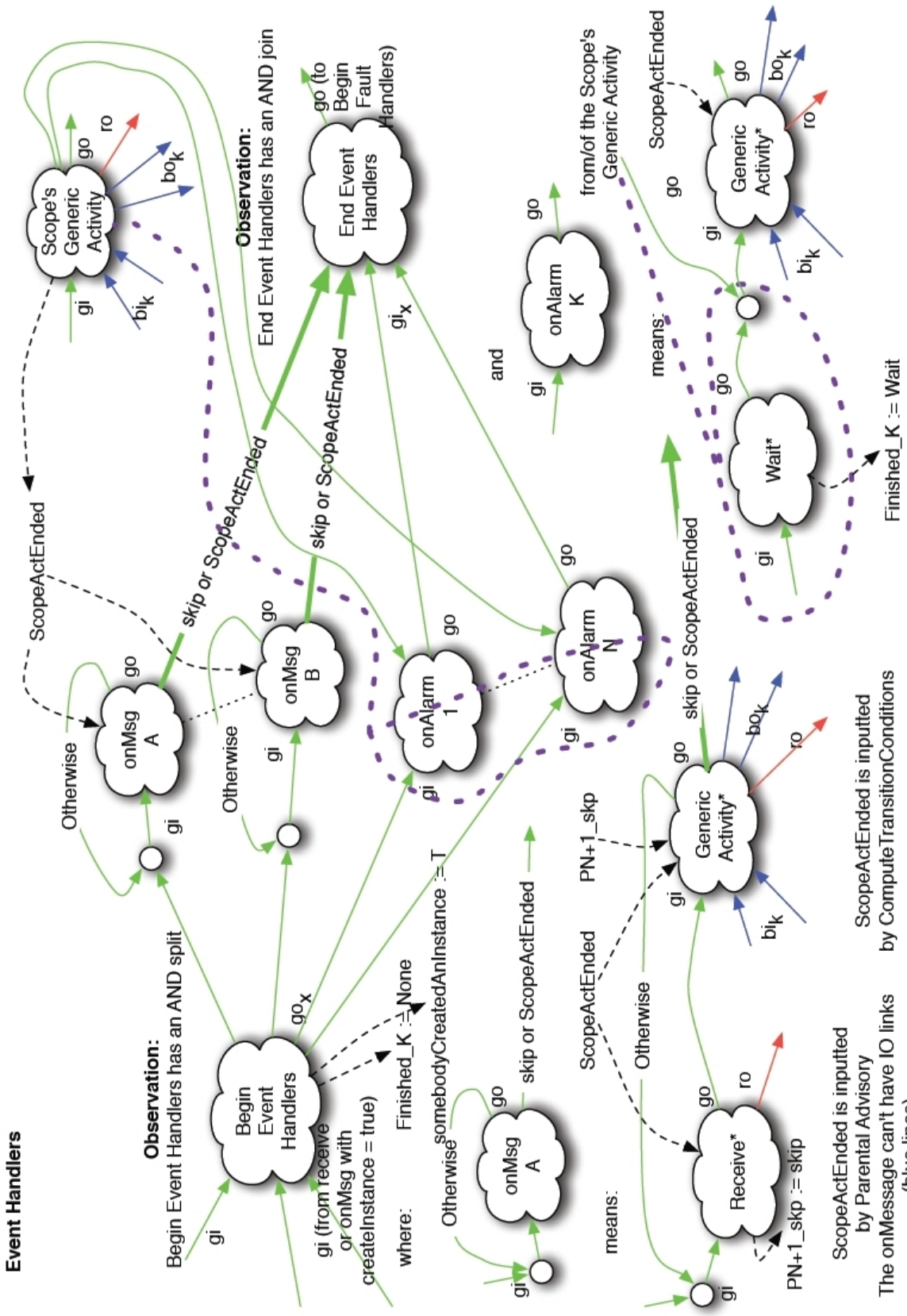
Compute Transition Conditions

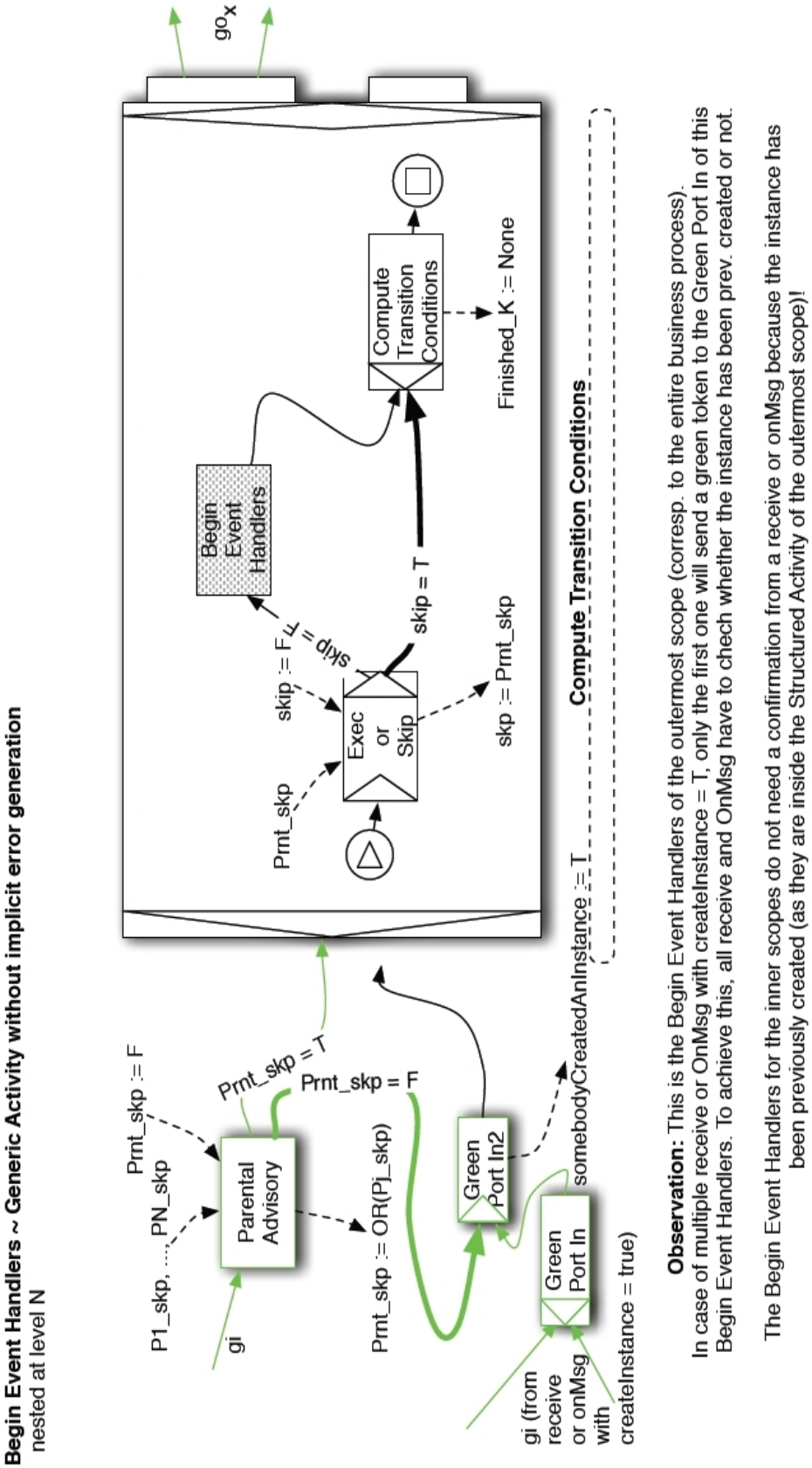
Observation: The green token is received in two cases:

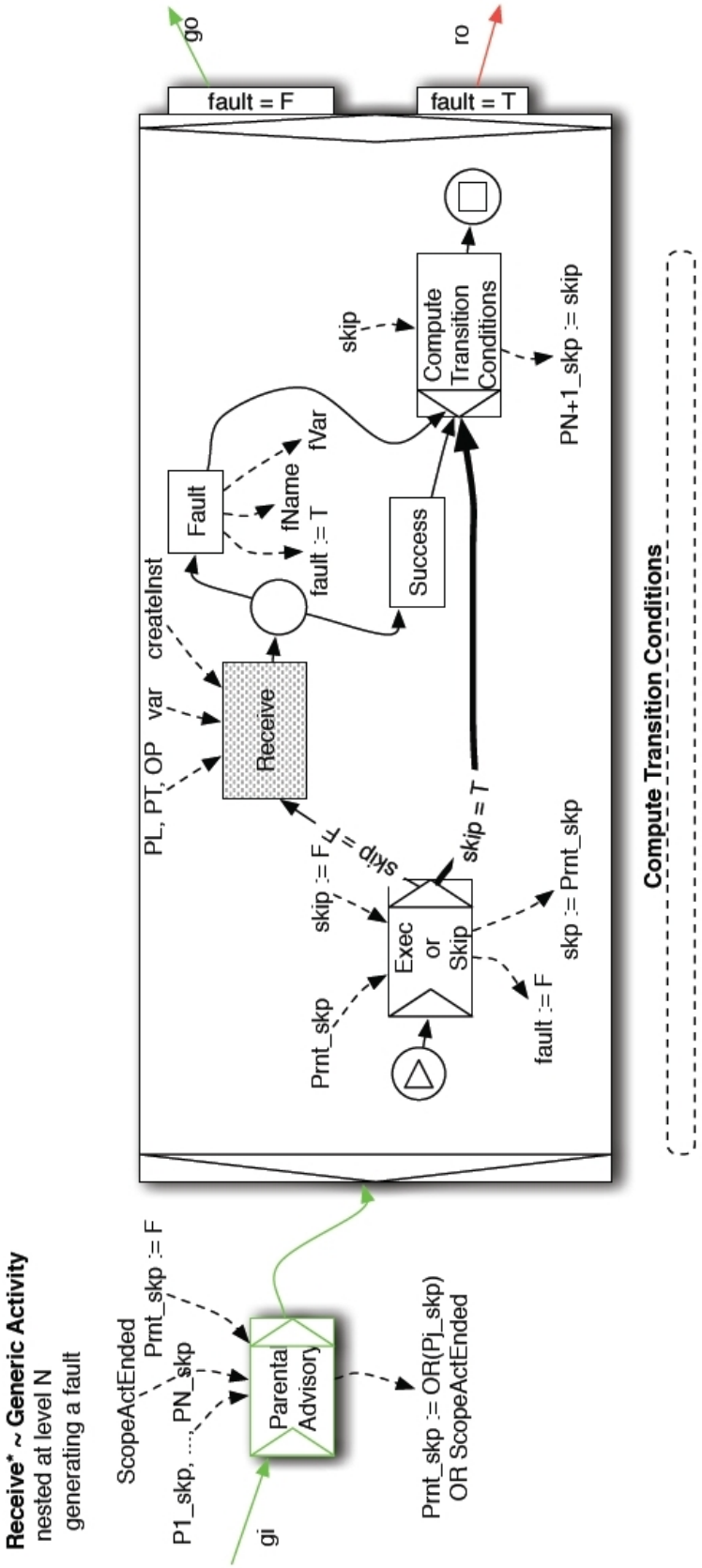
1. An error has been received and treated or not!
or,
2. The Fault Handlers had to be skipped -- either because the entire scope had to be skipped, or because the Scope's Activity was completed successfully!

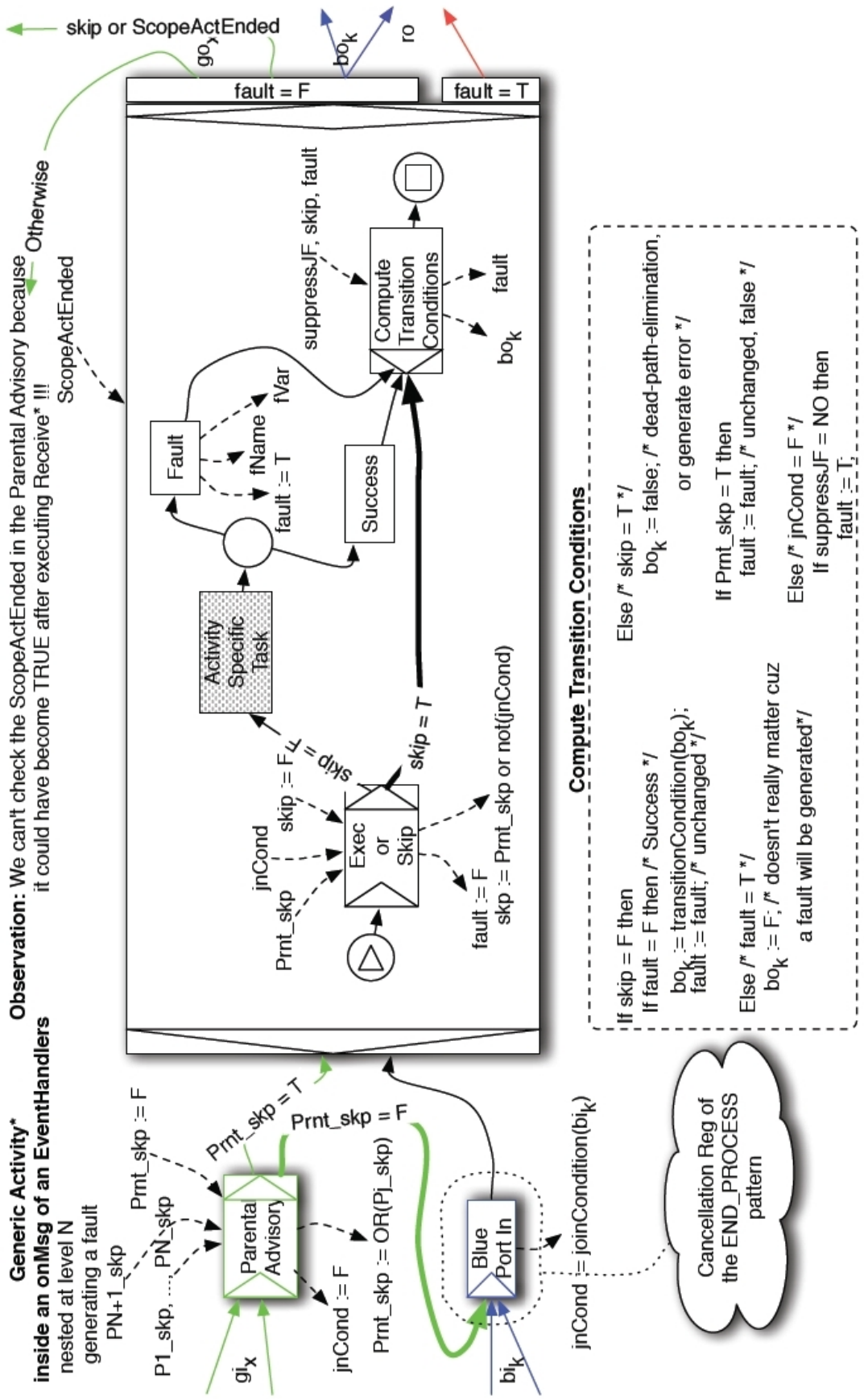
In the first case, the token is sent as End Scope will decide what to do (e.g., to throw the error further in case it was not processed (i.e., FAULT = YES)).
In the second case, a token will be sent to End Scope as well.

One should note that the Scope's Activity and the Fault Handlers are in a *SEQUENCE* as, anyway, "in the worst case", the activity completes its exec. and then we have to achieve the d-p-e inside the Fault Handlers.

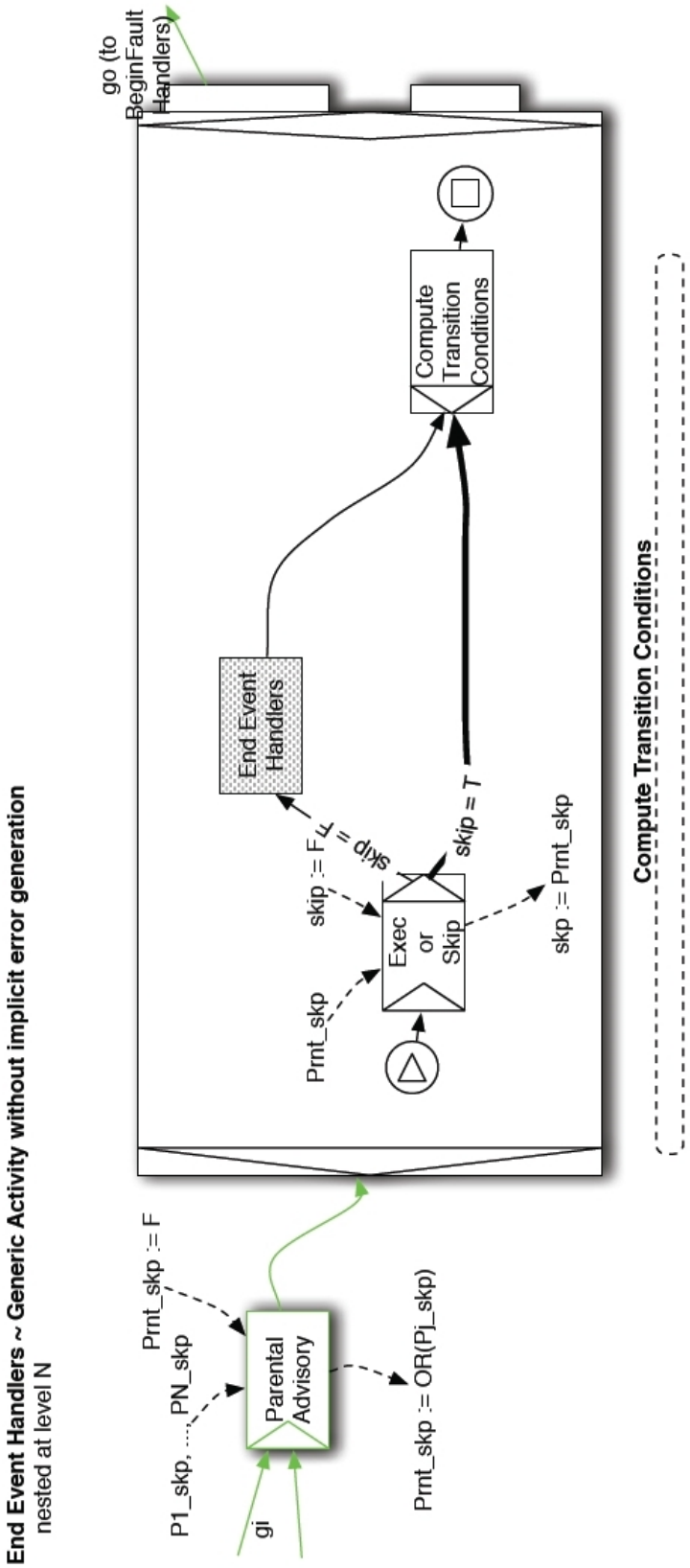

Event Handlers
Observation:
Begin Event Handlers has an AND split
Begin Event Handlers
gi (from receive or onMsg with createInstance = true)
where: Finished_K := None
Otherwise somebodyCreatedAnInstance := T
onMsg A
skip or ScopeActEnded
Otherwise
onMsg A
onMsg B
onAlarm 1
onAlarm N
onAlarm K
and
Scope's Generic Activity
ScopeActEnded
skip or ScopeActEnded
Observation:
End Event Handlers has an AND join
End Event Handlers
go (to Begin Fault Handlers)
from/of the Scope's Generic Activity
means:
ScopeActEnded
Generic Activity*
Wait*
Finished_K := Wait
means:
Otherwise
ScopeActEnded
PN+1_skp
Receive*
PN+1_skp := skip
ScopeActEnded is inputted by Parental Advisory
The onMessage can't have IO links (blue lines)
ScopeActEnded is inputted by ComputeTransitionConditions

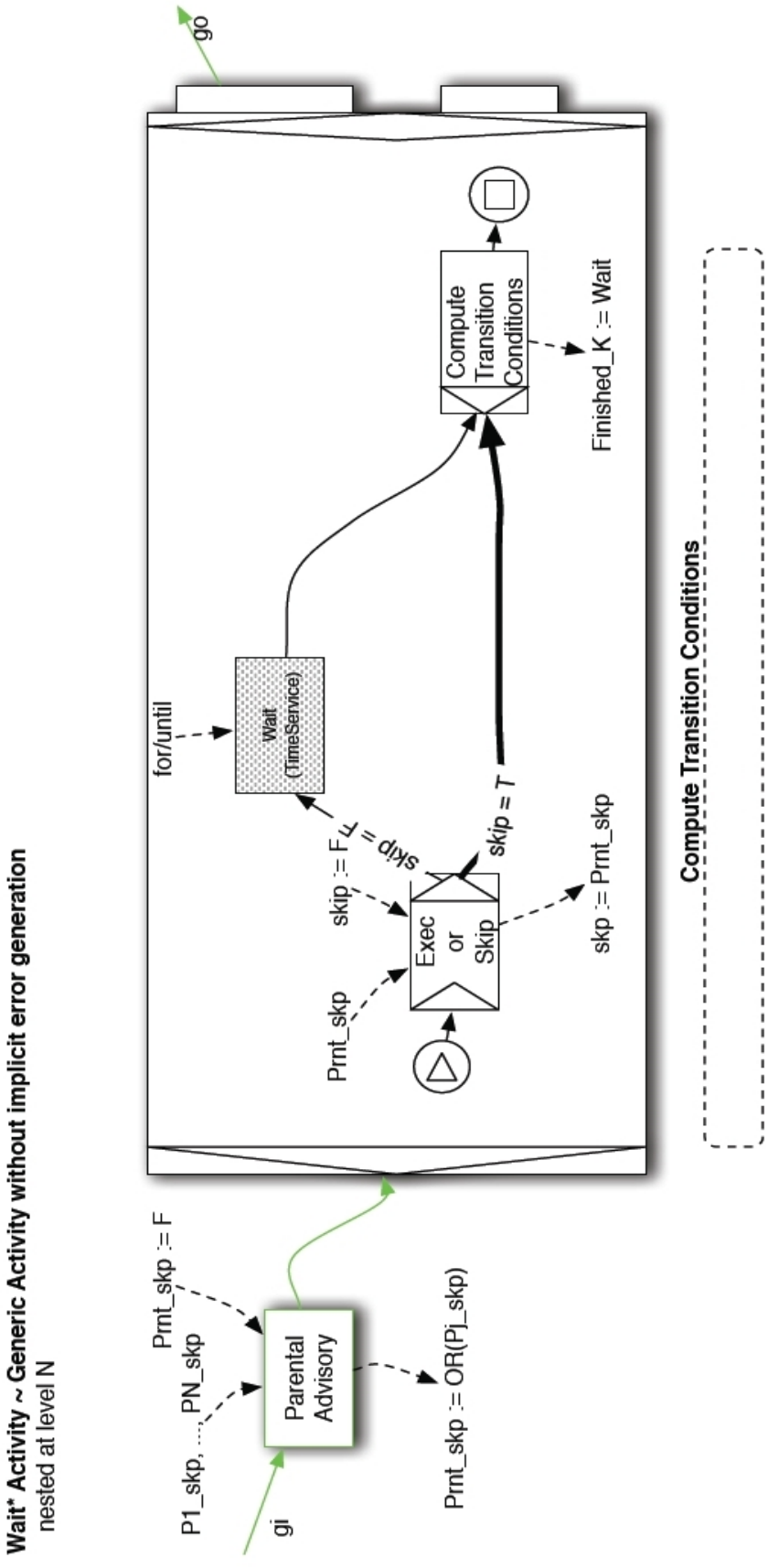

Begin Event Handlers ~ Generic Activity without implicit error generation
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
Prnt_skp := OR(Pi_skp)
Prnt_skp = T
Prnt_skp = F
Green Port In2
Green Port In
gi (from receive or onMsg with createInstance = true)
somebodyCreatedAnInstance := T
Compute Transition Conditions
Prnt_skp
Exec or Skip
skip := F
skip = T
skip = T
skp := Prnt_skp
Begin Event Handlers
Compute Transition Conditions
Finished_K := None
go_x
Observation: This is the Begin Event Handlers of the outermost scope (corresp. to the entire business process).
In case of multiple receive or OnMsg with createInstance = T, only the first one will send a green token to the Green Port In of this Begin Event Handlers. To achieve this, all receive and OnMsg have to chech whether the instance has been prev. created or not.
The Begin Event Handlers for the inner scopes do not need a confirmation from a receive or onMsg because the instance has been previously created (as they are inside the Structured Activity of the outermost scope)!

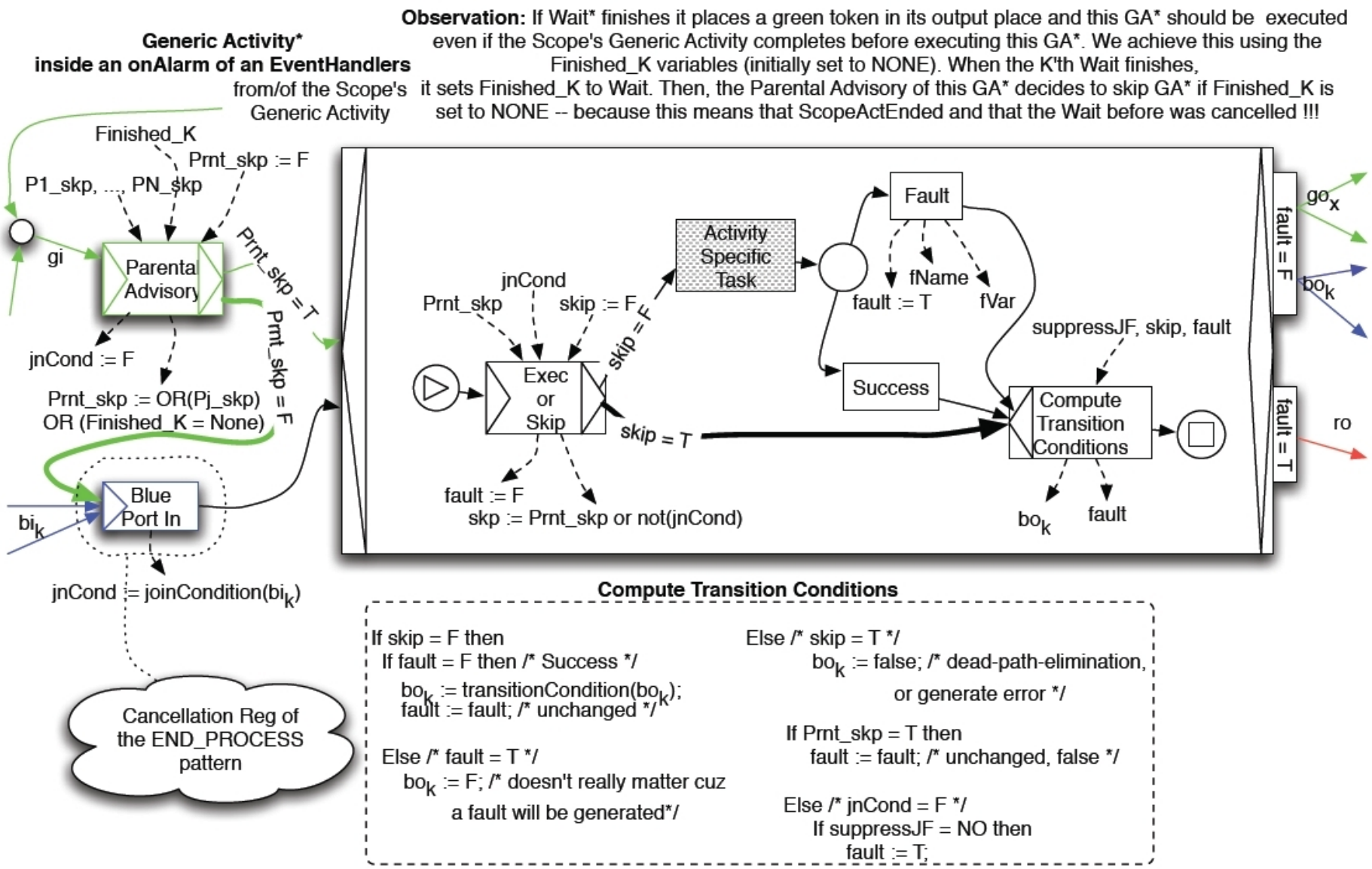

Receive* ~ Generic Activity
nested at level N
generating a fault
ScopeActEnded
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
Prnt_skp := OR(Pj_skp)
OR ScopeActEnded
go
ro
fault = F
fault = T
PL, PT, OP
var
createInst
Receive
skip := F
skip = T
Prnt_skp
Exec or Skip
skip := Prnt_skp
fault := F
Fault
fName
fVar
fault := T
Success
skip
Compute Transition Conditions
PN+1_skp := skip
Compute Transition Conditions

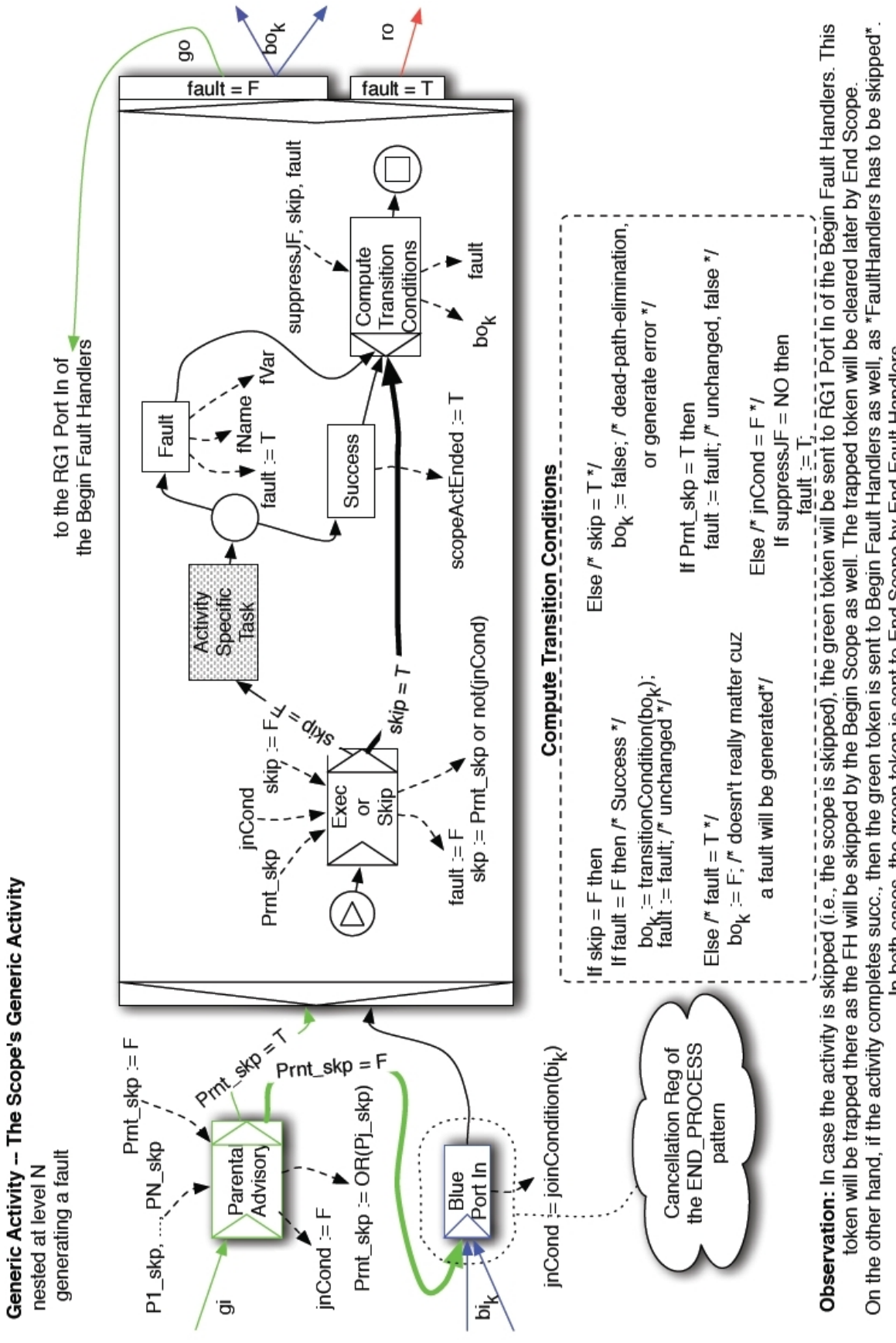


Observation: Check ScopeActEnded when sending the green token cuz maybe the scope ended while processing this onMessage. In this case, we go out of the loop so as to avoid a skipping cycle of this onMessage!

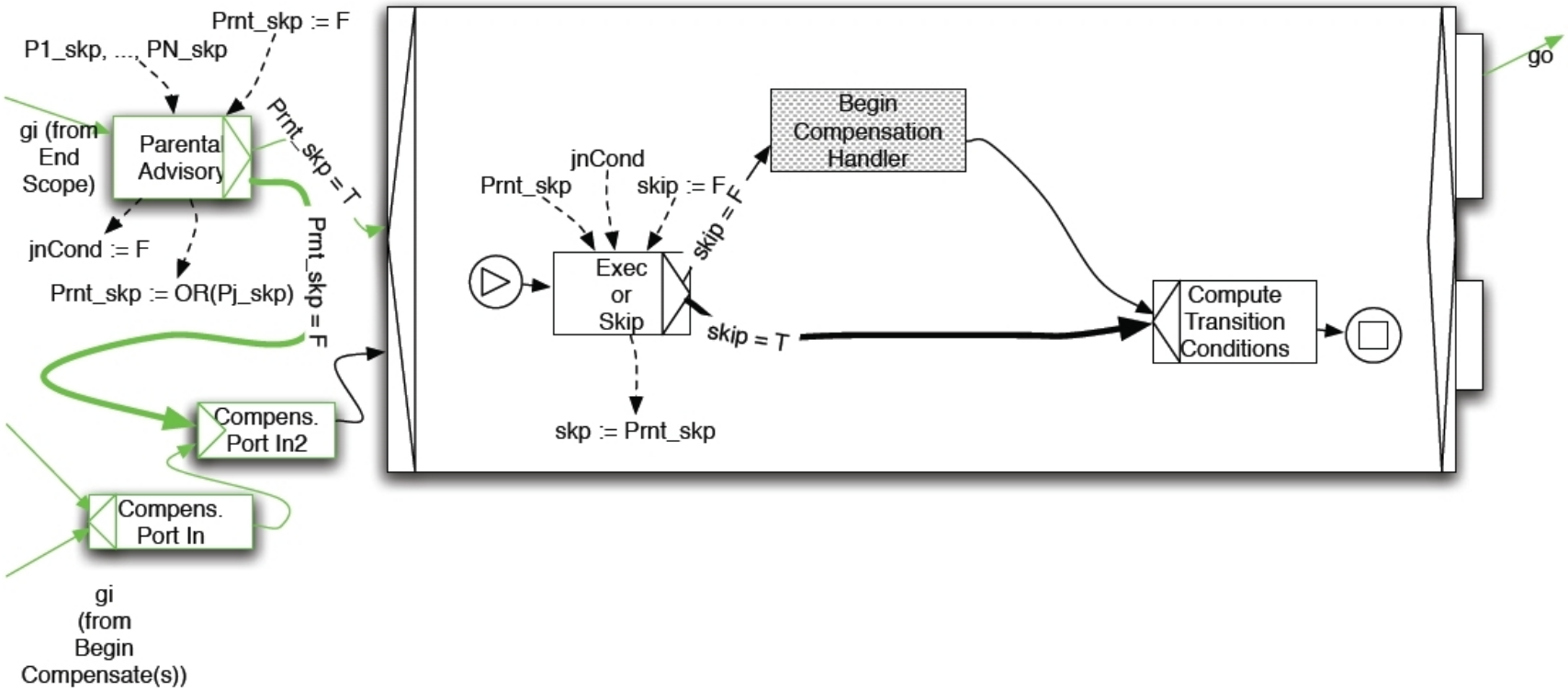

End Event Handlers ~ Generic Activity without implicit error generation
nested at level N
P1_skp, ..., PN_skp
Prnt_skp := F
gi
Parental Advisory
Prnt_skp := OR(Pj_skp)
go (to BeginFault Handlers)
Prnt_skp
skip := F
skip = T
Exec or Skip
skp := Prnt_skp
End Event Handlers
Compute Transition Conditions
Compute Transition Conditions


Wait* Activity ~ Generic Activity without implicit error generation
nested at level N
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
Prnt_skp := OR(Pj_skp)
go
for/until
Wait (TimeService)
skip := F
skip = F
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp
Compute Transition Conditions
Finished_K := Wait
Compute Transition Conditions


Generic Activity*
inside an onAlarm of an EventHandlers
from/of the Scope's Generic Activity
Observation: If Wait* finishes it places a green token in its output place and this GA* should be executed even if the Scope's Generic Activity completes before executing this GA*. We achieve this using the Finished_K variables (initially set to NONE). When the K'th Wait finishes, it sets Finished_K to Wait. Then, the Parental Advisory of this GA* decides to skip GA* if Finished_K is set to NONE -- because this means that ScopeActEnded and that the Wait before was cancelled !!!
Finished_K
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parental Advisory
Prnt_skp = T
Prnt_skp = F
jnCond := F
Prnt_skp := OR(Pj_skp) OR (Finished_K = None)
Blue Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of the END_PROCESS pattern
jnCond
Prnt_skp
skip := F
skip = F
Activity Specific Task
Fault
fName
fVar
fault := T
suppressJF, skip, fault
Exec or Skip
skip = T
Success
Compute Transition Conditions
fault := F
skp := Prnt_skp or not(jnCond)
bo_k
fault
fault = F
fault = T
go_x
bo_k
ro
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;

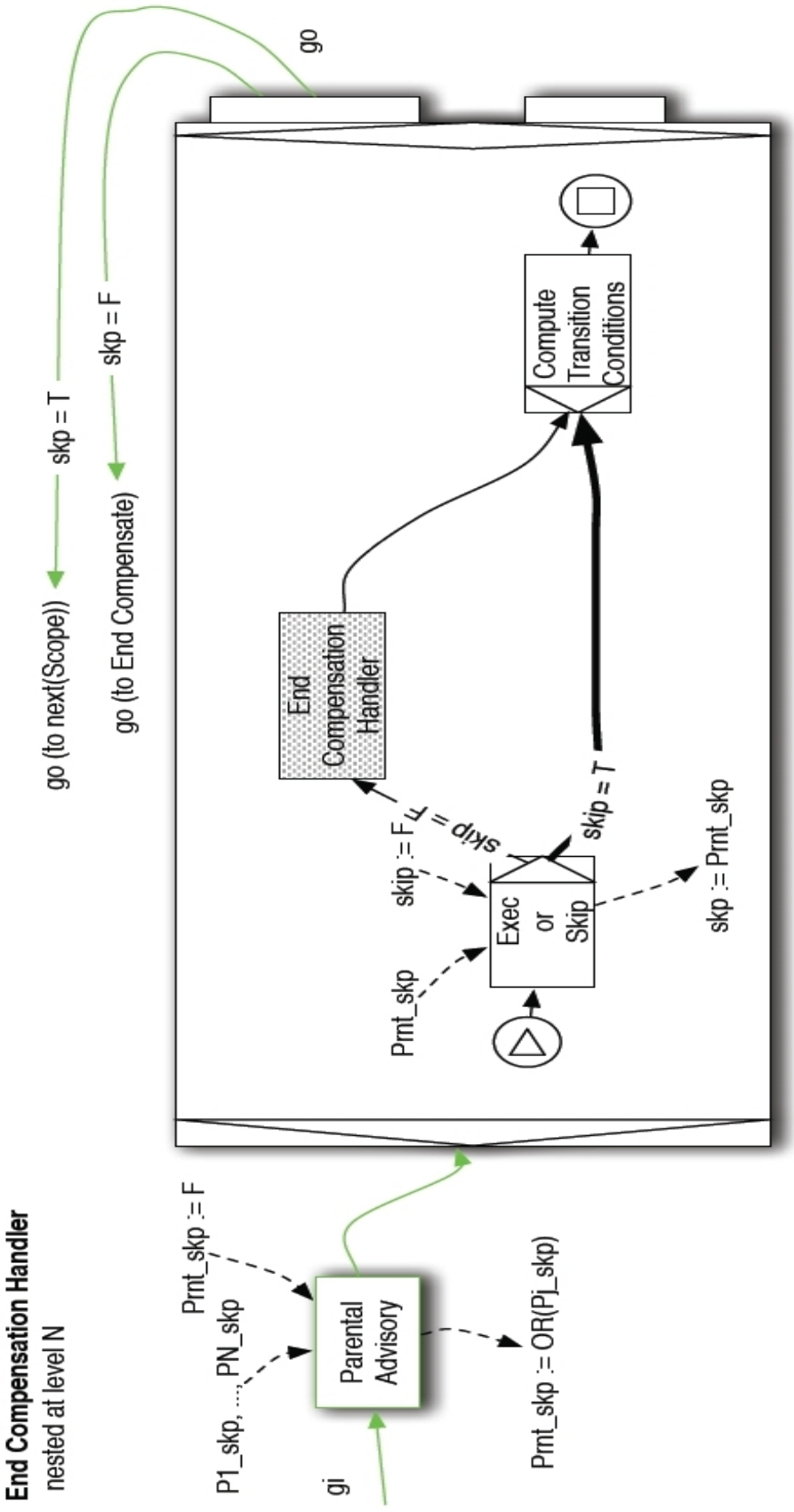

Generic Activity -- The Scope's Generic Activity
nested at level N
generating a fault
to the RG1 Port In of
the Begin Fault Handlers
go
bo_k
ro
fault = F
fault = T
suppressJF, skip, fault
Compute
Transition
Conditions
fault
bo_k
Fault
fName
fVar
fault := T
Success
scopeActEnded := T
Activity
Specific
Task
jnCond
skip := F
skip = T
Prnt_skp
Exec
or
Skip
fault := F
skp := Prnt_skp or not(jnCond)
Compute Transition Conditions
If skip = F then
If fault = F then /* Success */
bo_k := transitionCondition(bo_k);
fault := fault; /* unchanged */
Else /* fault = T */
bo_k := F; /* doesn't really matter cuz
a fault will be generated*/
Else /* skip = T */
bo_k := false; /* dead-path-elimination,
or generate error */
If Prnt_skp = T then
fault := fault; /* unchanged, false */
Else /* jnCond = F */
If suppressJF = NO then
fault := T;
Prnt_skp := F
P1_skp, ..., PN_skp
gi
Parenta
Advisor
jnCond := F
Prnt_skp := OR(Pj_skp)
Prnt_skp = T
Prnt_skp = F
Blue
Port In
bi_k
jnCond := joinCondition(bi_k)
Cancellation Reg of
the END_PROCESS
pattern
Observation: In case the activity is skipped (i.e., the scope is skipped), the green token will be sent to RG1 Port In of the Begin Fault Handlers. This
token will be trapped there as the FH will be skipped by the Begin Scope as well. The trapped token will be cleared later by End Scope.
On the other hand, if the activity completes succ., then the green token is sent to Begin Fault Handlers as well, as *FaultHandlers has to be skipped*.
In both cases, the green token is sent to End Scope by End Fault Handlers.

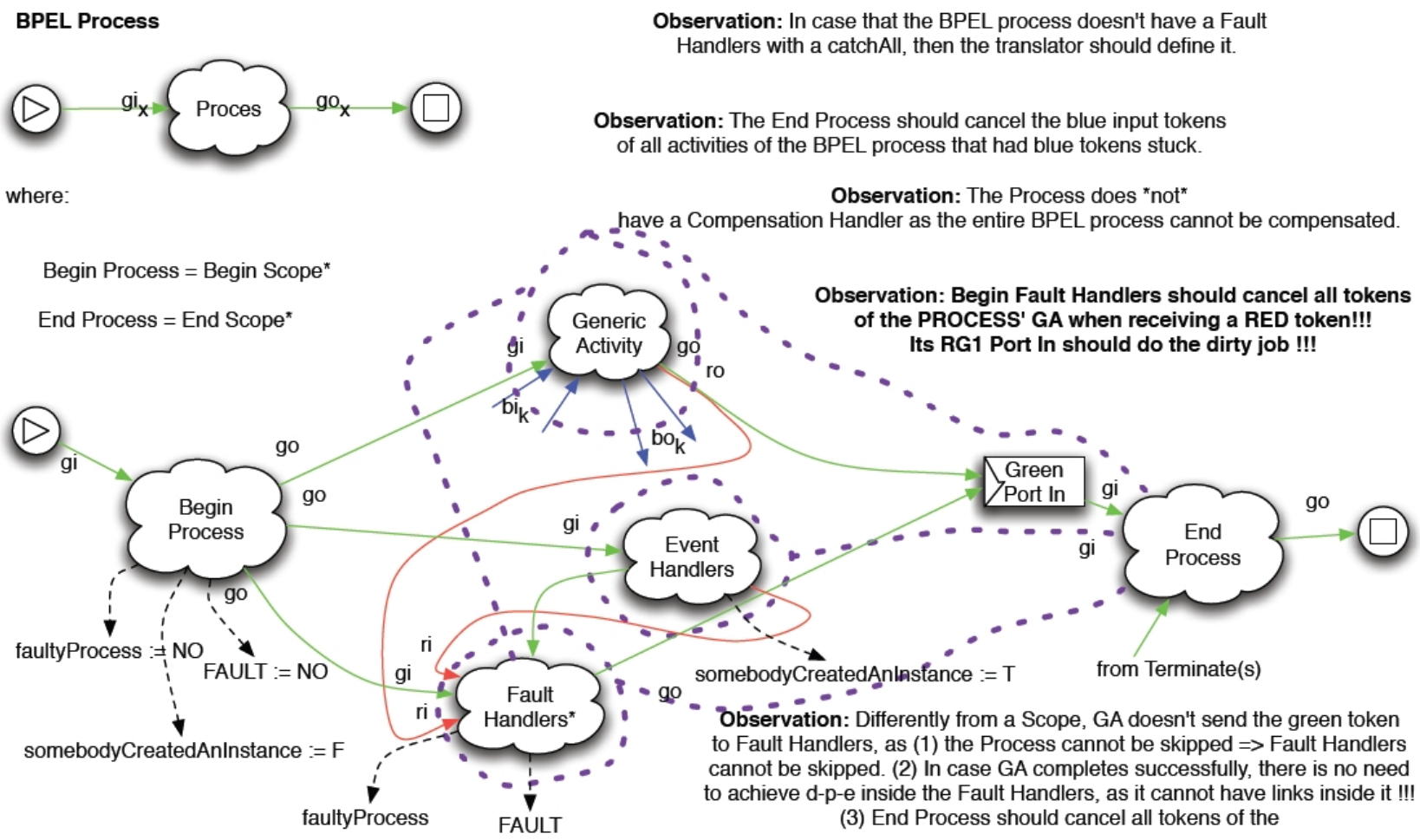
Begin Compensation Handler
nested at level N

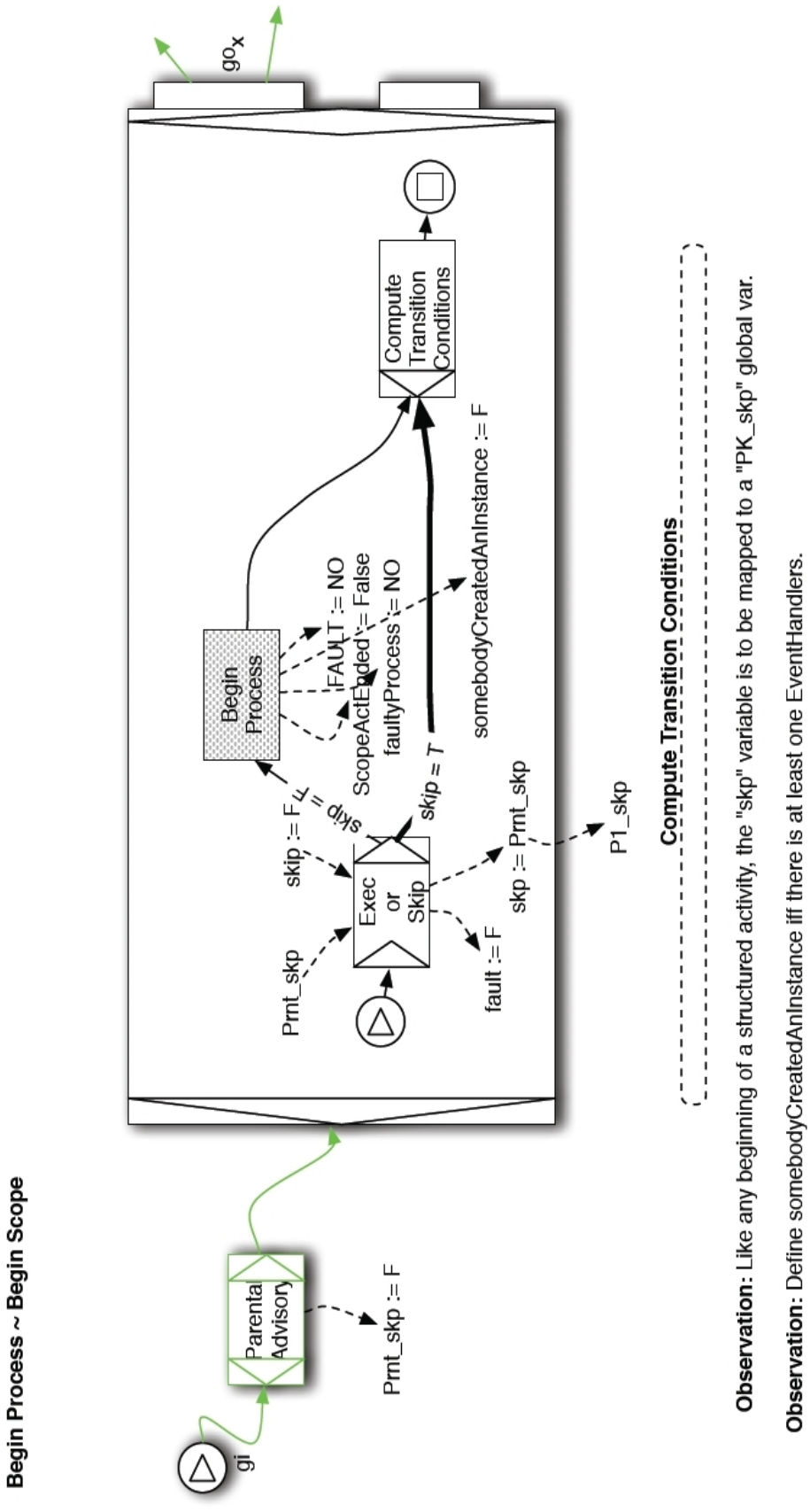


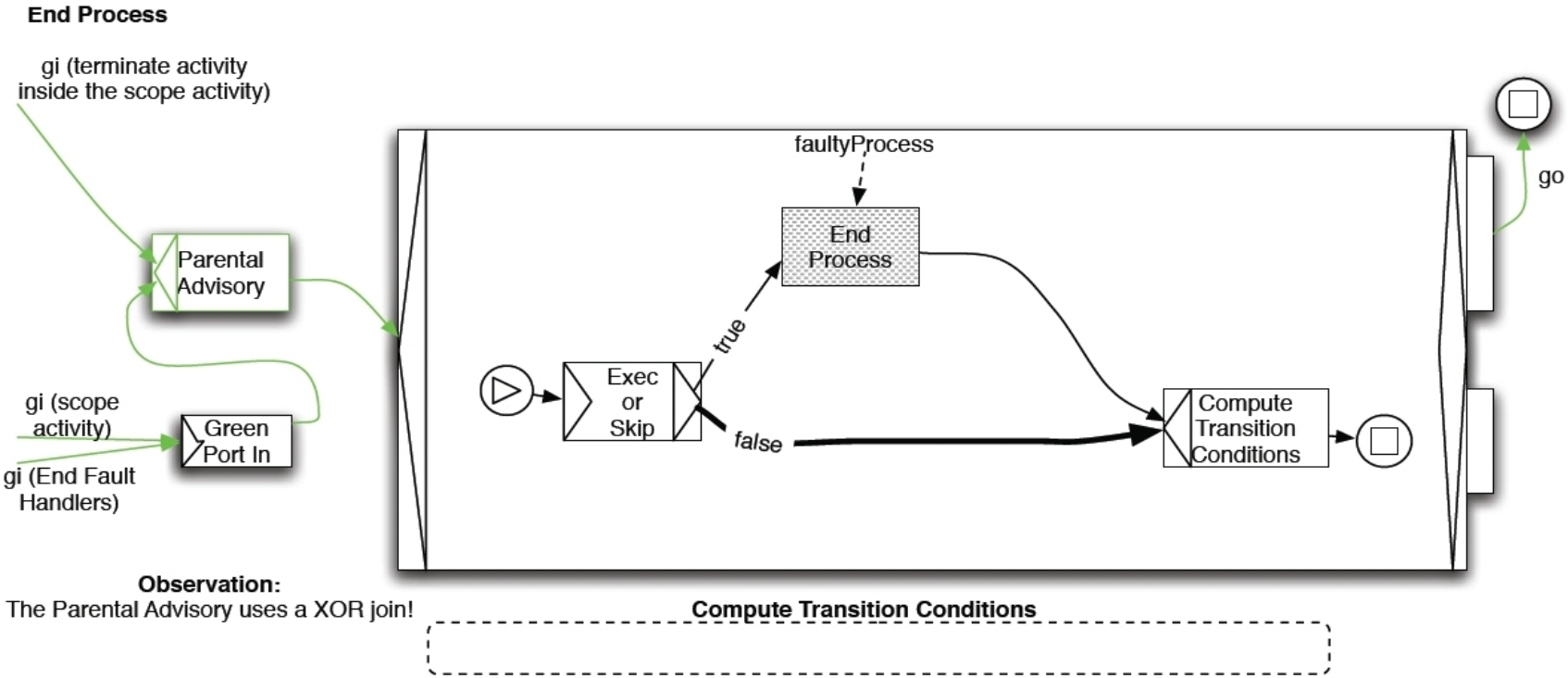
Observation: The Begin Compensate is in charge of checking whether the Scope to be compensated finished its execution (i.e., ScopeActEnded = T of the respective scope!).

Observation: The Generic Activity inside the Compensation handler is a normal Generic Activity.


End Compensation Handler
nested at level N
Prnt_skp := F
P1_skp, ...,PN_skp
gi
Parental Advisory
Prnt_skp := OR(Pi_skp)
go
go (to next(Scope))
skp = T
go (to End Compensate)
skp = F
End Compensation Handler
Compute Transition Conditions
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
skp := Prnt_skp


BPEL Process
Observation: In case that the BPEL process doesn't have a Fault Handlers with a catchAll, then the translator should define it.
gi_x
Proces
go_x
Observation: The End Process should cancel the blue input tokens of all activities of the BPEL process that had blue tokens stuck.
where:
Observation: The Process does *not* have a Compensation Handler as the entire BPEL process cannot be compensated.
Begin Process = Begin Scope*
End Process = End Scope*
Observation: Begin Fault Handlers should cancel all tokens of the PROCESS' GA when receiving a RED token!!! Its RG1 Port In should do the dirty job !!!
Generic Activity
gi
go
ro
bi_k
bo_k
gi
Begin Process
go
go
go
Event Handlers
gi
Green Port In
gi
End Process
go
gi
faultyProcess := NO
FAULT := NO
somebodyCreatedAnInstance := F
ri
gi
ri
Fault Handlers*
go
somebodyCreatedAnInstance := T
from Terminate(s)
Observation: Differently from a Scope, GA doesn't send the green token to Fault Handlers, as (1) the Process cannot be skipped => Fault Handlers cannot be skipped. (2) In case GA completes successfully, there is no need to achieve d-p-e inside the Fault Handlers, as it cannot have links inside it !!! (3) End Process should cancel all tokens of the
faultyProcess
FAULT
Observation: faultyProcess should be set to YES (inside the Fault Handlers of the process) in case FAULT=YES/OK!
somebodyCreatedAnInstance becomes true when a receive for e.g., sends a green token to the Event Handlers. It should be defined for the outermost Event Handlers only!
Observation: Fault Handlers* should define a "catchAll" with an "empty" activity in case that the BPEL process does not define it. We do so in order to catch all faults generated inside the business process.


Begin Process ~ Begin Scope
gi
Parenta Advisory
Prnt_skp := F
go_x
Compute Transition Conditions
Begin Process
FAULT := NO
ScopeActEnded := False
faultyProcess := NO
somebodyCreatedAnInstance := F
skip := F
skip = T
skip = T
Prnt_skp
Exec or Skip
fault := F
skp := Prnt_skp
P1_skp
Compute Transition Conditions
Observation: Like any beginning of a structured activity, the "skp" variable is to be mapped to a "PK_skp" global var.
Observation: Define somebodyCreatedAnInstance iff there is at least one EventHandlers.

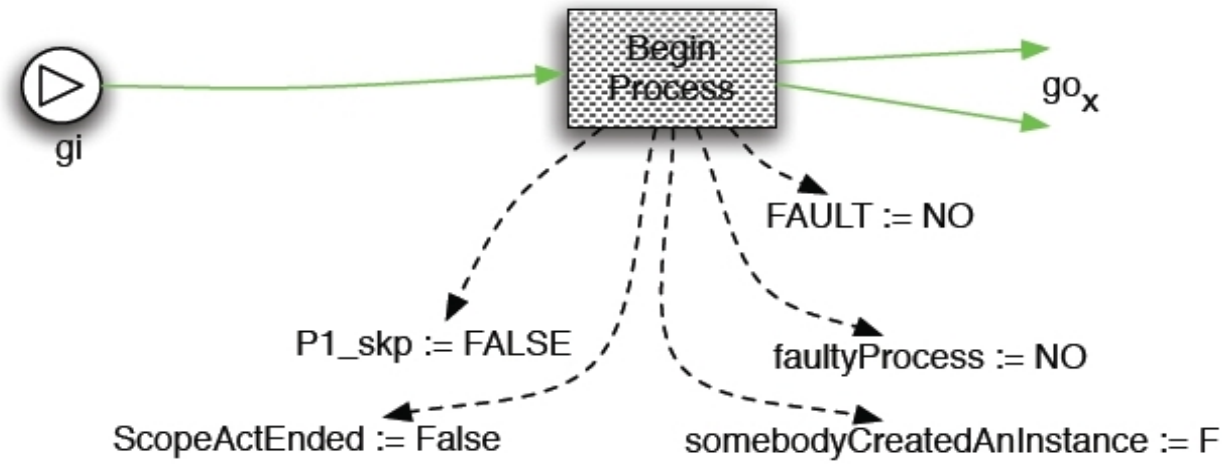


Observation: Input **faultyProcess** to see how it all went!

Observation: Each "terminate" activity should send the green token to End Process which will cancel all tokens.

Observation: We don't need to check whether a fault was caught or not cuz the (default) "catchAll" in the Fault Handlers will catch everything!

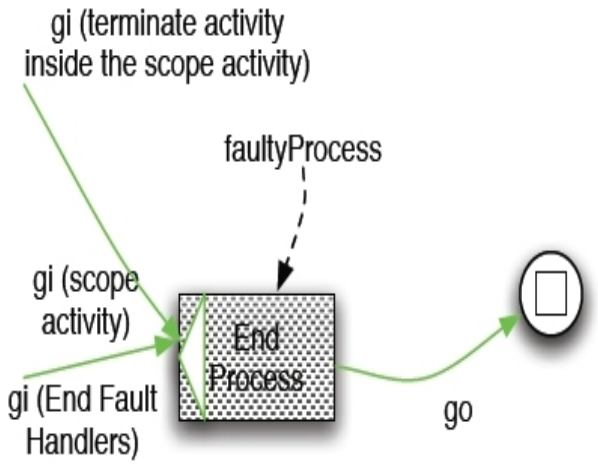
Begin Process -- SIMPLIFIED



Observation: Like any beginning of a structured activity, "PK_skp" is mapped a global var.

Observation: Define somebodyCreatedAnInstance iff there is at least one EventHandlers.

End Process -- SIMPLIFIED



Observation: Each "terminate" activity should send the green token to End Process which will cancel all tokens.

Observation: We don't need to check whether a fault was caught or not cuz the (default) "catchAll" in the Fault Handlers will catch everything!

Bibliografia

[1] – *Business Process Execution Language for Web Service version 1.1*, 5 maggio 2003
(http://www-128.ibm.com/developerworks/library/specification/ws-bpel/)

[2] – W.M.P. van der Aalst and A.H.M. ter Hofstede.
YAWL: Yet Another Workflow Language.
Information Systems, 30(4):245-275, 2005.

[3] – W.M.P. van der Aalst, L. Aldred, M. Dumas, A.H.M. ter Hofstede,
Design and Implementation of the Yawl System, giugno 2004
(http://www.yawl.fit.qut.edu.au/yawldocs/yawls.pdf)

[4] – W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski e A.P. Barros
Workflow Patterns, Luglio 2003
(http://is.tm.tue.nl/research/patterns/download/wfs-pat-2002.pdf)

[5] – Sean Kneipp e Lindsay Bradford
YAWL Editor 1.4 User Manual, febbraio 2006
(http://sky.fit.qut.edu.au/~terhofst/YAWLdocs/YAWLEditor1.4UserManual.pdf)

[6] – Christian Stahl
A Petri Net Semantics for BPEL
(http://edoc.hu-berlin.de/series/informatik-berichte/188/PDF/188.pdf)

[7] – W. M. P. van der Aalst
The Application of Petri Nets to Workflow Management, 1988
Journal of Circuits, Systems and Computers, 8(1):21--66, 1998.

[8] – Chun Ouyang, Eric Verbeek, W.M.P. van der Aalst, Stephen Breutel,

Marlon Dumas, and A.H.M. ter Hofstede

Formal Semantics and Analysis of Control Flow in WS-BPEL, 2005

(http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf)

[9] – M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, T. Newling

Patterns: Service-Oriented Architecture and Web Services, Redbooks, Aprile 2004

(http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf)

[10] – A. Brogi and R. Popescu.

Towards Semi-automated Workflow-Based Aggregation of Web Services.

In B. Benatallah, F. Casati, and P. Traverso, editors, Proceedings of ICSOC'05, volume 3826 of LNCS, pages 214–227. Springer, 2005.

[11] – A. Brogi and R. Popescu.

From BPEL Processes to YAWL Workflows. Technical Report,

Università di Pisa, Marzo 2006.

[12] – Papazoglou, M., and Georgakopoulos, D.

Service-oriented computing

Communications of the ACM 46(10):25–28, 2003

[13] – Jian Yang,

Web service componentization

Communications of the ACM 46(10): 35–40, 2003

[14] – Verbeek, E.,

WofYAWL. Technical report

Eindhoven University of Technolog, 2005

(http://home.tm.tue.nl/hverbeek/wofyawl03.pdf)

[15] – E. Christensen, F. Curbera, G. Meredith, S. Weerawarana,
Web Services Description Language (WSDL) 1.1, 15 Marzo 2001
(http://www.w3.org/TR/wsdl)

[16] – M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau,
SOAP Version 1.2 Part 1: Messaging Framework, 24 Giugno 2003
(http://www.w3.org/TR/soap12-part1/)

[17] – L. Clement, A. Hately, C. Von Riegen, T. Rogers
UDDI version 3.0.2, UDDI Spec Technical Committee Draft, 19 Ottobre 2004
(http://uddi.org/pubs/uddi_v3.htm)

[18] – X. Fu, T. Bultan, and J. Su.
Analysis of interacting BPEL web services.
In Proceedings of 13th International Conference on World Wide Web, pages 621-630,
New York, NY, USA, 2004. ACM Press.

[19] – H. Foster, S. Uchitel, J. Magee, and J. Kramer.
Model-based verification of Web service composition.
In Proceedings of 18th IEEE International Conference on Automated Software Engineering, pages 152-161, Montreal, Canada, October 2003. IEEE Computer Society.

[20] – J.A. Fisteus, L.S. Fernandez, and C.D. Kloos.
Formal verification of BPEL4WS business collaborations.
In Proceedings of 5th International Conference on Electronic Commerce and Web Technologies (EC-Web'04), volume 3180 of Lecture Notes in Computer Science, pages 76-85, Zaragoza, Spain, August 2004. Springer-Verlag.

[21] – A. Ferrara.
Web services: a process algebra approach.

In Proceedings of 2nd International Conference on Service Oriented Computing, pages 242-251, New York, NY, USA, 2004. ACM Press

[22] – M. Koshkina and F. van Breugel.
Verification of business processes for Web services.
Technical Report CS-2003-11, York University, October 2003.
(http://www.cs.yorku.ca/techreports/2003/CS-2003-11.ps)

[23] – R. Farahbod, U. GlÄasser, and M. Vajihollahi.
Abstract operational semantics of the Business Process Execution Language for Web
Services.
Technical Report SFU-CMPT-TR-2004-03, School of Computer Science, Simon Fraser University, Burnaby B.C. Canada, April 2004

[24] – A. Martens. Verteilte GeschÄaftsprozesse –
Modellierung und Verifikation mit Hilfe von Web Services (In German).
PhD thesis, Institut fÄur Informatik, Humboldt-UniversitÄat zu Berlin, Berlin, Germany, 2003.

[25] – S. Hinz, K. Schmidt, and C. Stahl.
Transforming BPEL to Petri nets
To appear in Proceedings of 3rd International Conference on Business Process Management, September 2005.

[26] – C. Stahl.
Transformation von BPEL4WS in Petrinetze (In German).
Master's thesis, Humboldt University, Berlin, Germany, 2004.

[27] – K. Schmidt and C. Stahl.
A Petri net semantic for BPEL.
In E. Kindler, editor, Proc. of 11th Workshop AWPN. Paderborn University, Oct. 2004.